The Linux Concept Journey

Version 5.0
April-2025

By Dr. Shlomi Boutnaru

Created using Craiyon, AI Image Generator

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. You can also find my other free eBooks at https://TheLearningJourney.com.

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

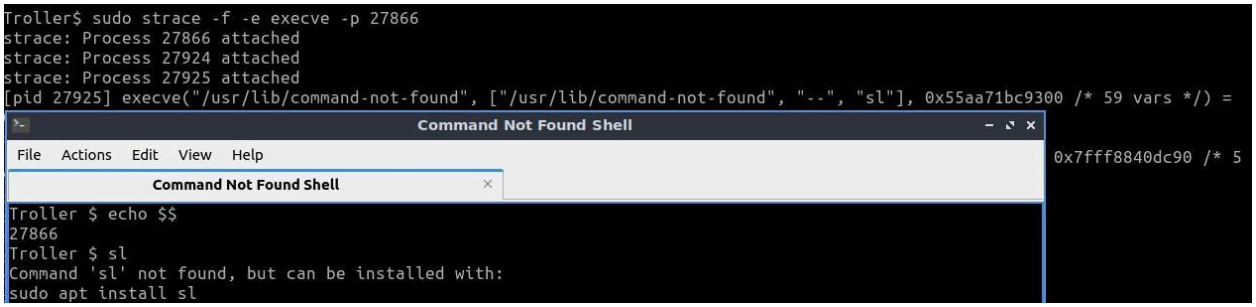
command not found

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).

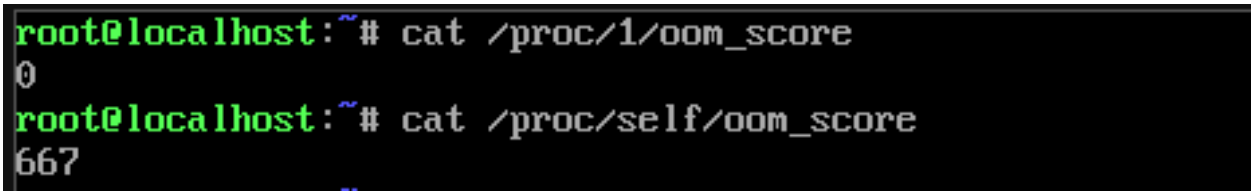


Out-of-Memory Killer (OOM killer)

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].



[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

```
Troller # cat code.c
#include <stdio.h>

void main()
{

        printf("\n**************!!!!!!!!\n");

}
Troller # gcc code.c -o code
Troller # ./code

**************!!!!!!!!
Troller # ltrace ./code

**************!!!!!!!!
+++ exited (status 24) +++
Troller # gcc -z lazy code.c -o ./code
Troller # ./code

**************!!!!!!!!
Troller # ltrace ./code
puts("\n**************!!!!!!!!"
**************!!!!!!!!
)                                                = 24
+++ exited (status 24) +++
Troller #
```

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

Syscalls (System Calls)

Syscalls (aka “System Calls”) are a fundamental interface between user-mode code and the Linux kernel. Most of the user-mode developers are not invoking syscalls directly, they are using wrappers as part of libraries (such as glibc). Those wrappers are very basic and mostly copy arguments to the right registers, calling the syscall and setting “errno” based on the return value from the system call. Today, there are more than 460 syscalls as part of Linux[8]. We can checkout the syscall numbers for different CPUs (“x86-64” and “arm 32”, “arm 64” and “x86”) as part of “Chromium OS Docs”[9]. For watching which syscalls are called by a binary we can use “strace”[10] - as shown in the screenshot below (taken using https://copy.sh/v86/?profile=archlinux).

Overall, in order to implement a syscall as part of the Linux kernel source code the “SYSCALL_DEFINEn” macro is used (where “n” is the number of arguments needed by the syscall implementation like “SYSCALL_DEFINE3”). By the way, this macro is based on the “SYSCALL_METADATA” and “SYSCALL_DEFINEx” macros[11]. We can see that as part of “/include/linux/syscalls.h”[12] - more on that in future writeups.

Lastly, the way in which arguments are passed to the kernel for a syscall is based on an ABI (Application Binary Interface) per CPU architecture. The ABI is composed of: the relevant instruction for calling the syscall, the register which holds the number of the syscall, register/s which is going to hold the return value of the syscall, a register that can hold an error value and a list of registers for passing the arguments (up to 7) for the syscall. Let us take as an example “x86-64”: the instruction is “syscall”, “rax” holds the number of the syscall, “rax+rdx” holds the return value and “rdi, rsi ,rdx, r10, r8 and r9” used for the arguments[13].

[8] https://man7.org/linux/man-pages/man2/syscalls.2.html
[9] https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md
[10] https://man7.org/linux/man-pages/man1/strace.1.html
[11] https://lwn.net/Articles/604287/
[12] https://elixir.bootlin.com/linux/v6.12.4/source/include/linux/syscalls.h#L216
[13] https://man7.org/linux/man-pages/man2/syscall.2.html

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[14].

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[15].

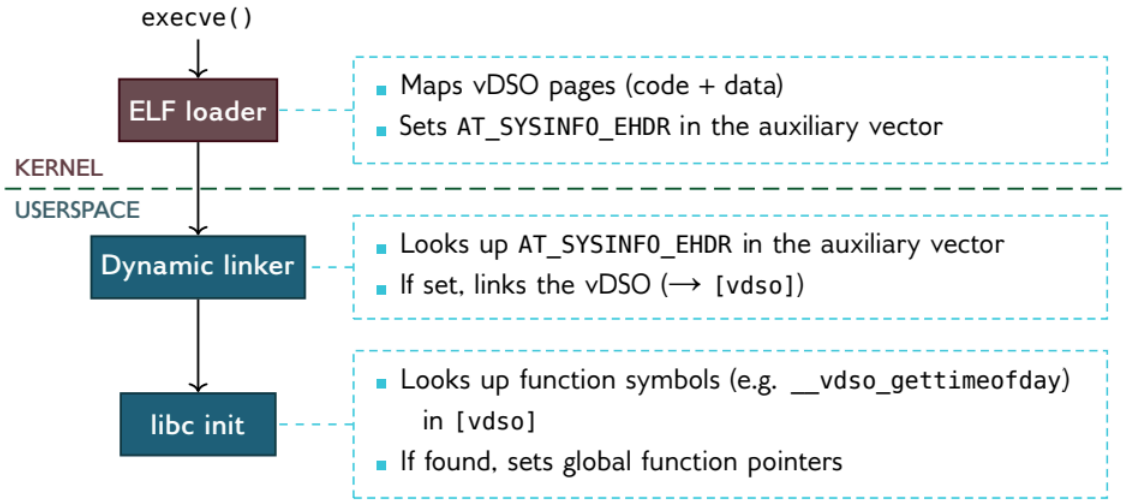
If you vDSO the implementation of the syscall interface is defined by the kernel in the following manner. A set of CPU instructions formatted as ELF is mapped to the end of the user-mode address space of all processes - as shown in the screenshot below. When libc needs to execute a syscall it checks 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[16].

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).

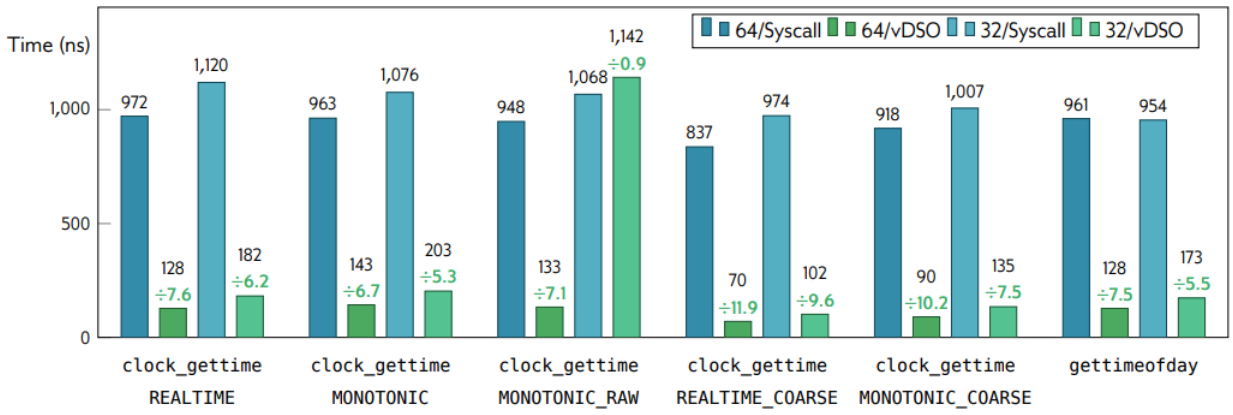
Also, as part of the kernel compilation process the vDSO code is compiled and linked. We can most of the time find it using the following command “find arch/$ARCH/ -name '*vdso*.so*' -o -name '*gate*.so*'”[17]
If we want to enable/disable vDSO we can set “/proc/sys/vm/vdso_enable” to 1/0 respectively[18].
Lastly, a benchmark of different syscalls using different implementations is shown below.

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

Kernel and userspace setup



ARM



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'[19] 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
```

[19] 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()[20].

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.

[20] 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]
```

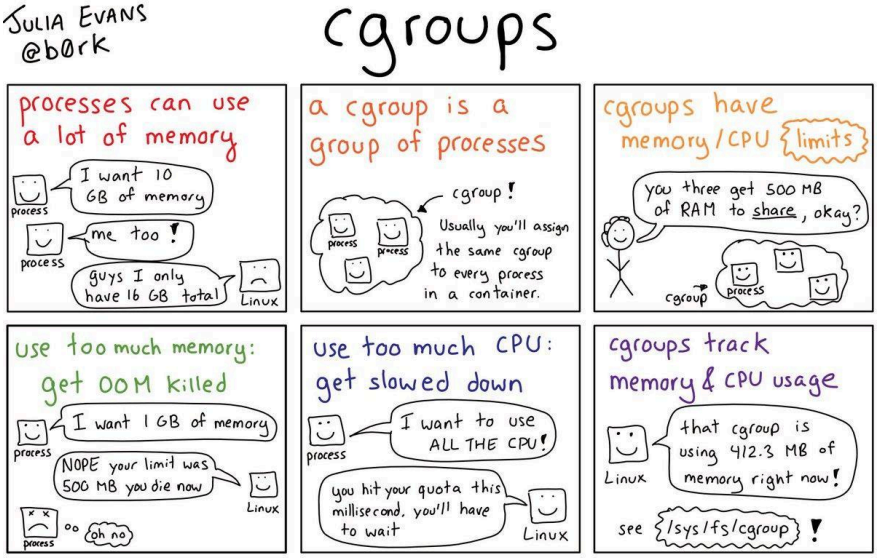
cgroups (Control Groups)

“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[21].

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[22]. The documentation for cgroups v2 first appeared in the Linux kernel 4.5 release on March 2016[23].

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[24]. By the way, since kernel 4.19 OOM killer[25] is aware of cgroups, which means the OS can kill a cgroup as a single unit.



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

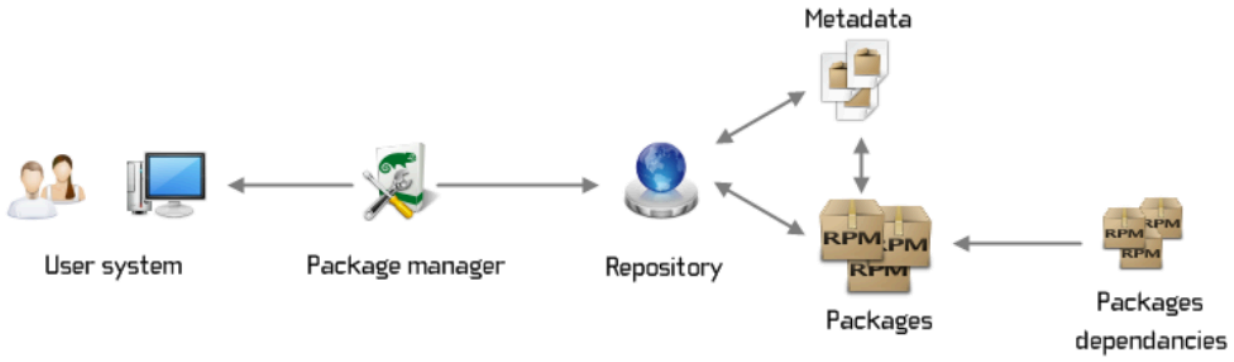
Package Managers

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) ?

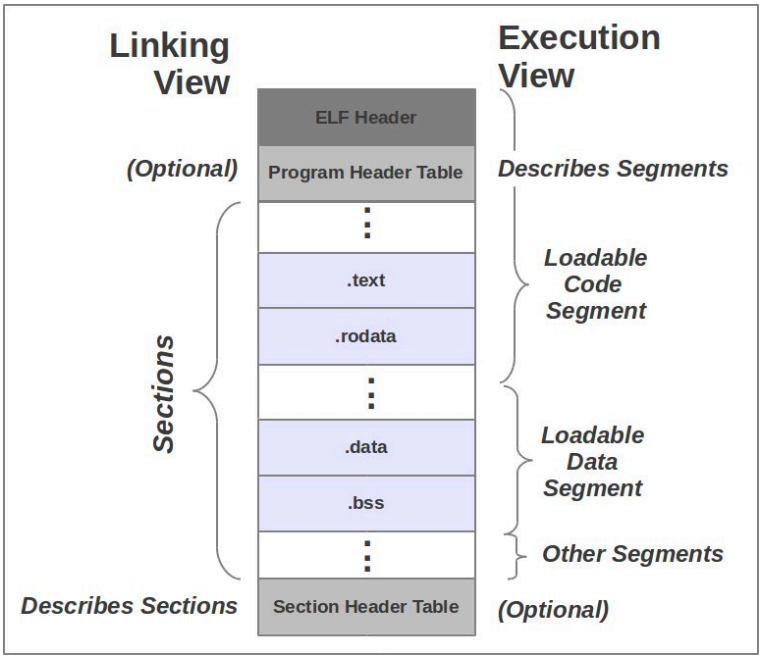
Every generic/standard operating system has a binary format for its user mode executables/libraries, kernel code and more. Windows has PE (Portable Executable), OSX has MachO and Linux has ELF. We are going to start with ELF (I promise to go over the others also).

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[26].

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”).



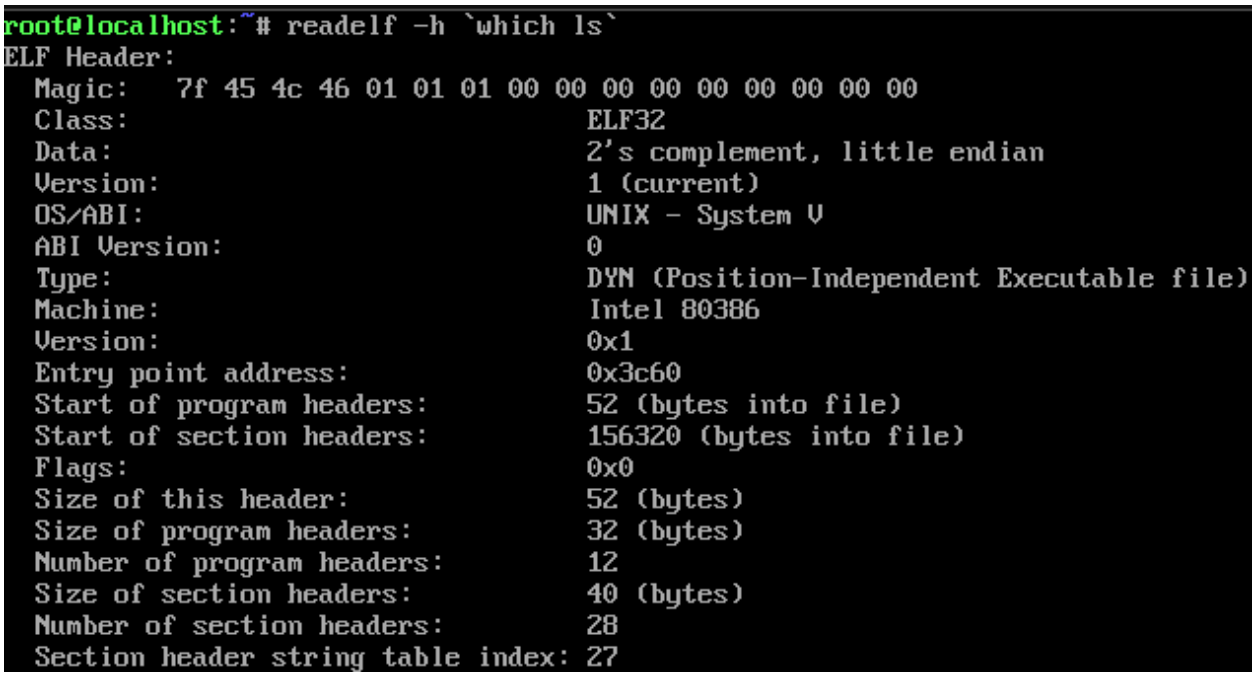
[26] https://i.stack.imgur.com/RMV0g.png

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).



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[27]. 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

```
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
```

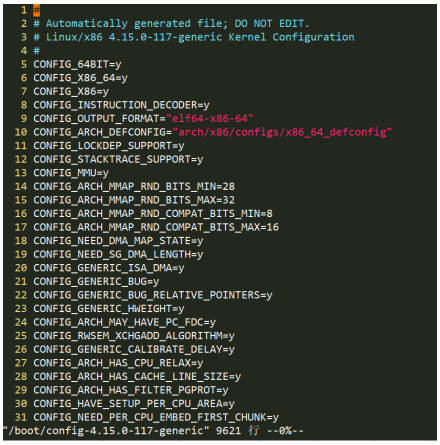
[27] https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/usr.html

/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[28]. 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”[29].

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[30]. The configuration file is based on key values - as shown in the screenshot below[31]. 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[32]. Also, we can adjust a specific value of features like the “CONFIG_ARCH_MMAP_RND_BITS_MIN=28”[33].

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[34]. Thus, if “CONFIG_DRM_TTM=m” then the “TTM memory manager subsystem” is going to be compiled outside of the kernel[35]. If “ttm” is loaded it will be shown in the output of “lsmod”[36].



[28] https://linux.die.net/man/1/uname
[29] https://unix.stackexchange.com/questions/123026/where-kernel-configuration-file-is-stored
[30] https://linuxconfig.org/in-depth-howto-on-linux-kernel-configuration
[31] https://blog.csdn.net/weixin_43644245/article/details/121578858
[32] https://elixir.bootlin.com/linux/v6.4.11/source/arch/um/Kconfig#L36
[33] https://elixir.bootlin.com/linux/v6.4.11/source/arch/x86/Kconfig#L322
[34] https://stackoverflow.com/questions/14587251/understanding-boot-config-file
[35] https://github.com/torvalds/linux/blob/master/drivers/gpu/drm/ttm/ttm_module.c
[36] 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"[37].

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[38] or zgrep[39] 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[40] - as also shown in the screenshot below. We can also go overt the creation of the "/proc" entry[41] and the function that returns the data when reading that entry[42].

```
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
```

[37] https://medium.com/@boutnaru/the-linux-concept-journey-boot-config-uname-r-6a4dd16048c4
[38] https://linux.die.net/man/1/zcat
[39] https://linux.die.net/man/1/zgrep
[40] https://elixir.bootlin.com/linux/v6.5/source/kernel/configs.c#L35
[41] https://elixir.bootlin.com/linux/v6.5/source/kernel/configs.c#L60
[42] https://elixir.bootlin.com/linux/v6.5/source/kernel/configs.c#L41

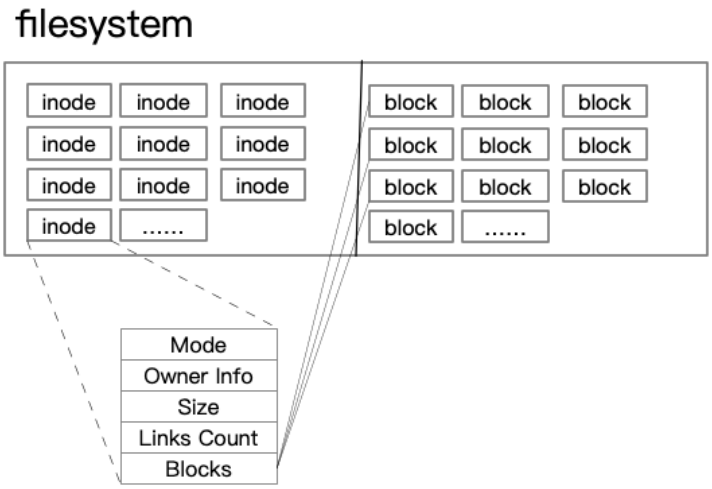
What is an inode?

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[43]. An illustration of that is shown below[44].

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[45].

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"[46]. Also, we can see the inode of a file/directory and other metadata of the file using the command "ls -i"[47] or "stat"[48]. By the way, the "stat" command can use different syscalls (depending on the filesystem and the specific version) like "stat"[49], "lstat"[50] or "statx"[51].

Lastly, you can check out "struct inode" in the source code of the Linux kernel[52]. Not all the points/links are directly connected to the data blocks, however I will elaborate on that in a future writeup.



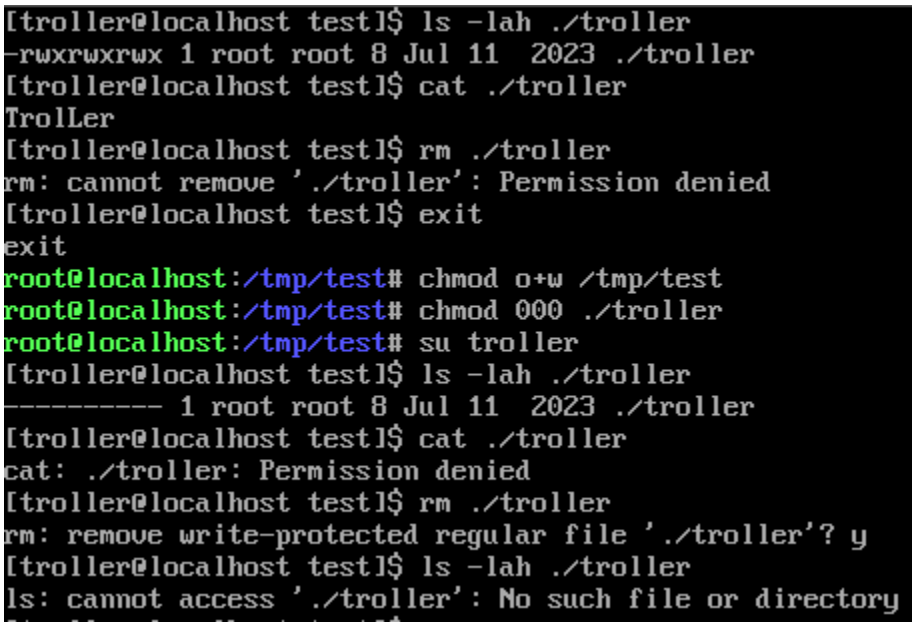
[43] https://www.bluematador.com/blog/what-is-an-inode-and-what-are-they-used-for
[44] https://www.sobyte.net/post/2022-05/linux-inode/
[45] https://www.stackscale.com/blog/inodes-linux/
[46] https://linux.die.net/man/1/df
[47] https://man7.org/linux/man-pages/man1/ls.1.html
[48] https://linux.die.net/man/1/stat
[49] https://linux.die.net/man/2/stat
[50] https://linux.die.net/man/2/lstat
[51] https://man7.org/linux/man-pages/man2/statx.2.html
[52] 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[53] or the "unlinkat" syscall[54].

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[55] 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.



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

VFS (Virtual File System)

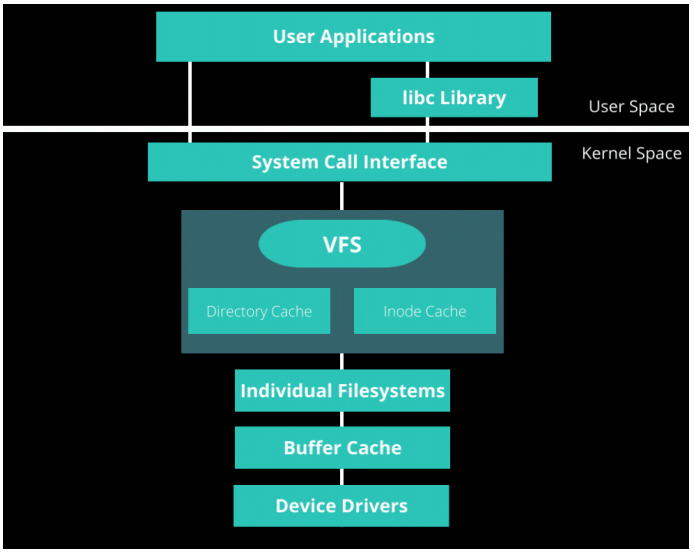
VFS (Virtual File System, aka Virtual File Switch) is a software component of Linux which is responsible for the filesystem interface between the user-mode and kernel mode. Using it allows the kernel to provide an abstraction layer that makes implementation of different filesystems very easy[56].

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[57].

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[58].

Thus, "inode"[59] 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").



[56] https://www.kernel.org/doc/html/next/filesystems/vfs.html
[57] https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-high-level-tour
[58] https://www.win.tue.nl/~aeb/linux/lk/lk-8.html
[59] 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[60]. By the way, it used to be called “shm fs”[61].

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[62]. 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[63]. We can see the implementation as part of the Linux’s kernel source code[64].

```
root@localhost:/tmp# mount | grep troller
root@localhost:/tmp# mount -t tmpfs tmpfs -o size=1M /tmp/troller
root@localhost:/tmp# mount | grep troller
tmpfs on /tmp/troller type tmpfs (rw,relatime,size=1024k)
root@localhost:/tmp# dd if=/dev/zero of=/tmp/troller/tmp bs=512 count=9999999999
dd: error writing '/tmp/troller/tmp': No space left on device
2049+0 records in
2048+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0801 s, 13.1 MB/s
root@localhost:/tmp# echo test > /tmp/troller/test
-bash: echo: write error: No space left on device
```

[60] https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html
[61] https://cateee.net/lkddb/web-lkddb/TMPFS.html
[62] https://man7.org/linux/man-pages/man5/tmpfs.5.html
[63] https://cateee.net/lkddb/web-lkddb/TMPFS.html
[64] https://elixir.bootlin.com/linux/v6.6-rc1/source/mm/shmem.c#L133

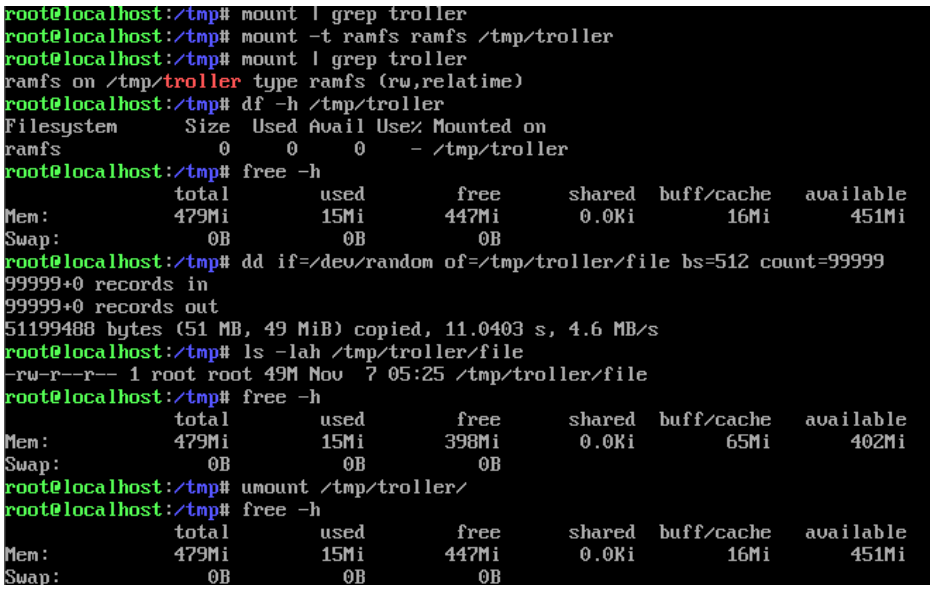
ramfs (Random Access Memory Filesystem)

“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[65].

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[66].

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”[67] is that “tmpfs” is limited in size and can also be swapped[68].

Lastly, we can go over the implementation of “ramfs” as part of Linux's kernel source code[69]. There are two implementations, one in case of an MMU[70] and one in case there is no MMU[71]. A good example for using “ramfs” is “initramfs”.



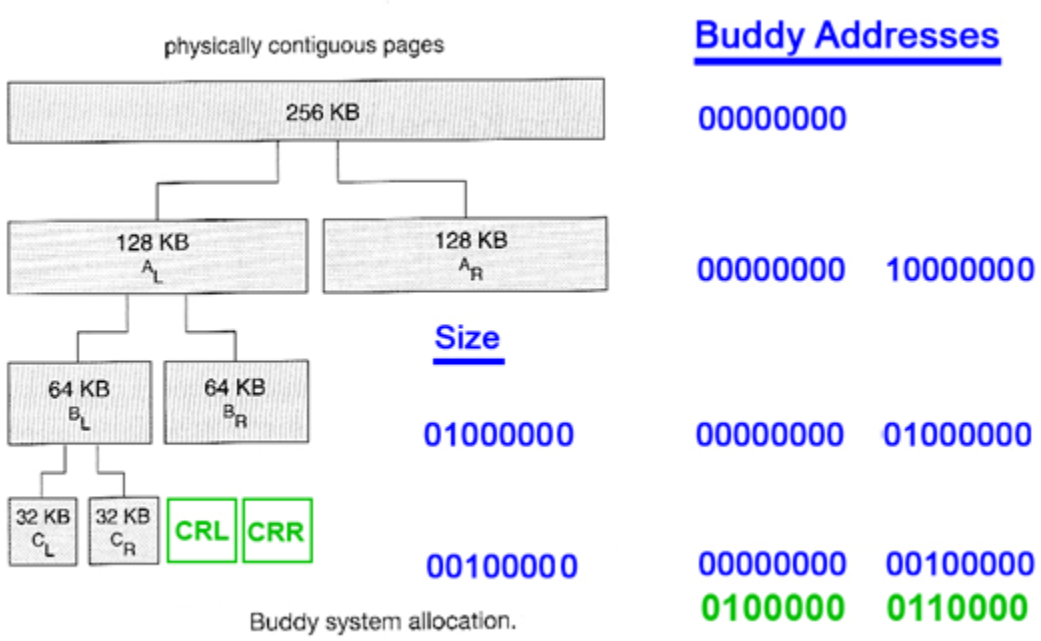
[65] https://docs.kernel.org/filesystems/ramfs-rootfs-initramfs.html
[66] https://lwn.net/Articles/157676/
[67] https://medium.com/@boutnaru/the-linux-concept-journey-tmpfs-temporary-filesystem-886b61a545a0
[68] https://wiki.debian.org/ramfs
[69] https://elixir.bootlin.com/linux/v6.5.3/source/fs/ramfs
[70] https://elixir.bootlin.com/linux/v6.5.3/source/fs/ramfs/file-mmu.c
[71] 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[72]

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[73].

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[74]. 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.

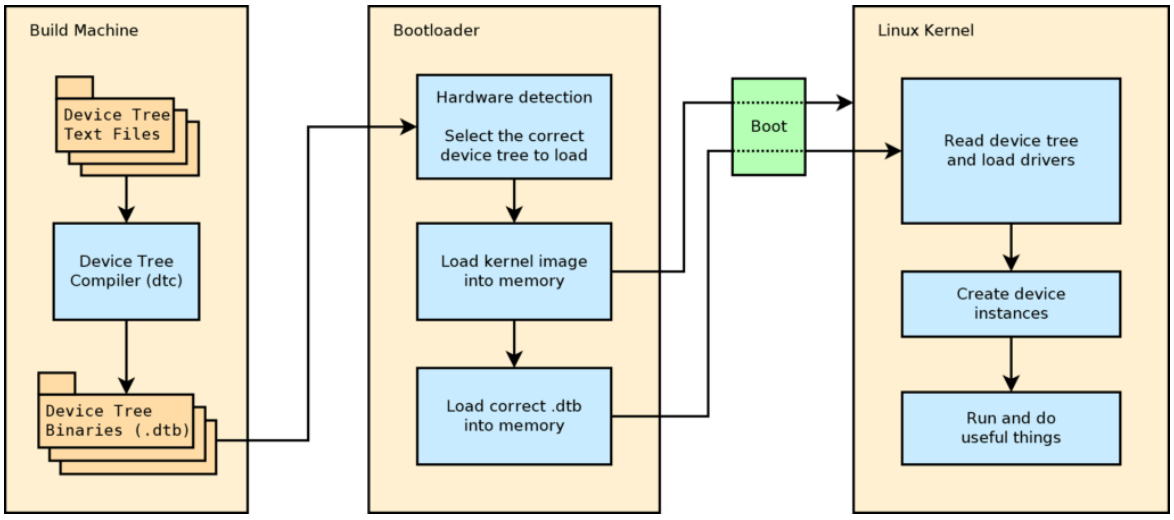
[72] https://en.wikipedia.org/wiki/Buddy_memory_allocation
[73] https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/
[74] https://www.expertsmind.com/questions/describe-the-buddy-system-of-memory-allocation-3019462.aspx

DeviceTree

Overall, a “Device Tree” is a mechanism for describing non-discoverable hardware. In the past this type of information was hard-coded as part of the source code (and later part of the binary/firmware). So we can say a “DeviceTree” is a data structure for describing hardware. By the way, there is a specification for that called “The Devicetree Specification”[75].

Moreover, “DeviceTree” allows the kernel to manage different components such as CPU, memory, buses and other. The specification detailed above contains the data format used, which is internally a tree of named nodes and properties (key-value based). Device trees can be in binary format (“*.dtb”) or text based (“*.dts”) for easing the management and editing[76]. The “*.dtb” files are loaded by the bootloader and passed to the kernel - as shown in the diagram below[77].

Thus, we can find in the Linux kernel source code “*.dts” files in the directories of some of the architectures, with the following pattern “/arch/[ARCHITECTURE/boot/dts”. Two examples are “arm64”[78] and “nios2”[79]. In order to transform “*.dts” files to “*.tdb” we can use the “dtc” which is the “Device Tree Compiler”[80]. We can also go over the Makefiles responsible for that if we want[81].
Lastly, there are also “.dtsi” files that include files (like we have in c/c++). We can use also YAML as the format of “*.dts” files in order to describe the different hardware components[82].



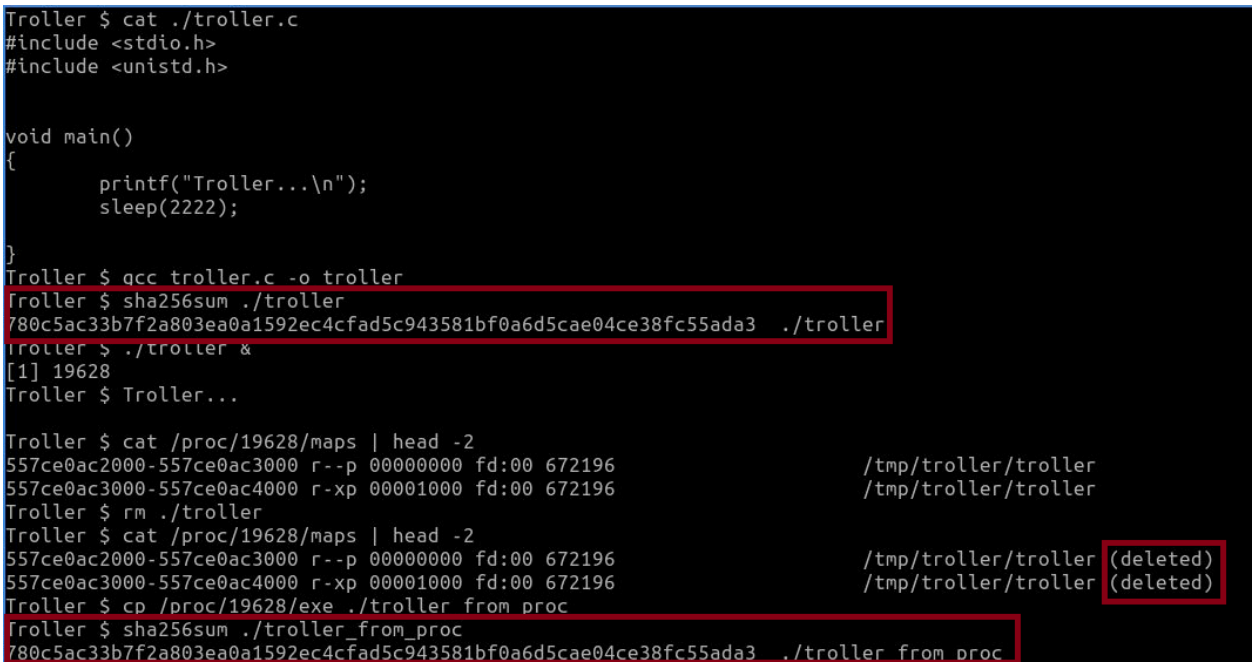
[75] https://www.devicetree.org/specifications/
[76] https://en.wikipedia.org/wiki/Devicetree
[77] https://vocal.com/resources/development/what-is-linux-device-tree/
[78] https://elixir.bootlin.com/linux/v6.5.4/source/arch/arm64/boot/dts
[79] https://elixir.bootlin.com/linux/v6.5.4/source/arch/nios2/boot/dts
[80] https://manpages.ubuntu.com/manpages/xenial/man1/dtc.1.html
[81] https://elixir.bootlin.com/linux/v6.5.4/source/arch/arm64/boot/dts/Makefile
[82] https://www.konsulko.com/yaml-and-device-tree

How can we recover a deleted executable of a running application?

In contrast to what you may think in case an executable file is deleted there are cases in which we can recover it very easily. One of them is in case there is at least one instance of an application running which is based on that executable.

Moreover, if the file is deleted we will see an indication for that in "/proc/[PID]/maps", the string pattern "(deleted)" is added in that case - as shown in the screenshot below. Thus, we can recover the file by copying it from the location "/proc/[PID]/exe" - as also shown in the screenshot below.

Lastly, this can be used for forensics/security purposes or just in case we deleted the executable by mistake (just remember not to stop/kill the application). By the way, we can use a similar trick for other files which are not the main image/executable (but that is for future writeups).



Process Group

Overall, a “Process Group” is a collection of processes which can be managed/handled together by the operating system (one example of that is signal management). Each “Process Group” has an identifier (PGID) and a “leader” which is the process that has created it. The PGID is equal to the PID of the group leader[83].

Moreover, we can use getpgid/getpgrp to get PGID or setpgid/setpgrp to set it[84]. The setgrp/getgrp is a System-V API which setpgid/getpgid is the POSIX API[85]. The POSIX one is the preferred one. Thus, if we check grep.app we can see that “setgpid”[86] has more than 29 times more results than “setpgrp”[87].

Lastly, any time we execute a command in a shell or a pipeline of commands we create a “Process Group” (it is sometimes called a job in the shell). As we can see in the screenshot below the number of the “Process Group” changes for each execution, and for the last one it gets incremented by one only even though we execute 5 processes (cause it is the same pipeline).

```
root@localhost:/tmp/troller# cat print_pgid.py
import ctypes
libc=ctypes.CDLL(None)
print(libc.getpgid(libc.getpid())) #we can also pass 0 to getgpid(), instead of using getpid()

root@localhost:/tmp/troller# python ./print_pgid.py
1281
root@localhost:/tmp/troller# python ./print_pgid.py
1282
root@localhost:/tmp/troller# ps | ps | ps | ps | python ./print_pgid.py
1283
```

[83] https://biriukov.dev/docs/fd-pipe-session-terminal/3-process-groups-jobs-and-sessions/
[84] https://man7.org/linux/man-pages/man2/setpgid.2.html
[85] https://unix.stackexchange.com/questions/404054/how-is-a-process-group-id-set
[86] https://grep.app/search?q=%20setpgid%28
[87] https://grep.app/search?q=%20setpgrp%28

<Major:Minor> Numbers

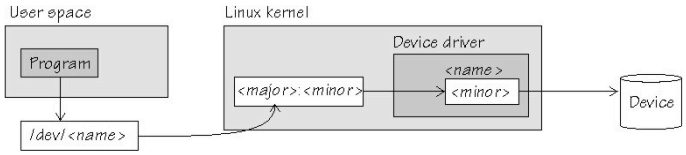
In the Linux kernel devices (character/block device) are represented as a pair of numbers (<major>:<minor>). There are some major/minor numbers which are reserved while others are assigned dynamically. A major number can be shared between multiple device drivers[88].

Overall, device files are located in "/dev" and they allow accessing the device from user-mode. Those files are "connected" to the device using the major and minor number - as shown in the diagram below on the left side[89]. We can see them using "ls -l" - as shown in the shell output in the screenshot below (taken from copy.sh). The major number identifies the drivers associated with the device, while the minor number is used only by the driver which gets the number without the kernel using it[90].

Moreover, the kernel code that can assign the name and the major number for a specific device char device can use the "register_chrdev" function[91]. In case of a block device can use the macro "register_blkdev"[92].

Lastly, we can use the "mknod"[93] command to create a block/char device file by providing a name, major and minor numbers. We can also list the major numbers of the currently registered devices with their names[94] - as shown in the output of the screenshot below.

```
root@localhost:/dev# ls -la /dev/null /dev/random /dev/urandom /dev/zero
crw-rw-rw- 1 root root 1, 3 Nov  7 02:50 /dev/null
crw-rw-rw- 1 root root 1, 8 Nov  7 02:50 /dev/random
crw-rw-rw- 1 root root 1, 9 Nov  7 02:50 /dev/urandom
crw-rw-rw- 1 root root 1, 5 Nov  7 02:50 /dev/zero
root@localhost:/dev# cat /proc/devices | head -10
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
```



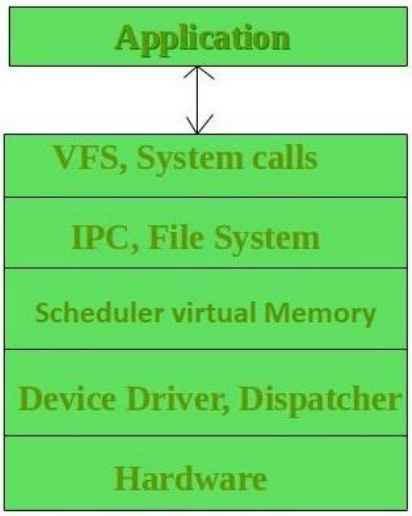
[88] https://www.ibm.com/docs/en/linux-on-systems?topic=hdaa-device-nodes-numbers
[89] https://www.ibm.com/docs/en/linuxonibm/com.ibm.linux.z.ufdd/lxnode.jpg
[90] https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch03s02.html
[91] https://elixir.bootlin.com/linux/v6.6.1/source/include/linux/fs.h#L2535
[92] https://elixir.bootlin.com/linux/v6.6.1/source/include/linux/blkdev.h#L809
[93] https://man7.org/linux/man-pages/man1/mknod.1.html
[94] https://man7.org/linux/man-pages/man5/procfs.5.html

Monolithic Kernel

Monolithic kernel is an operating system architecture in which the entire OS code is executed in kernel mode. Examples of operating systems which use this design are: Linux, BSD, SunOS, AIX, MS-DOS, OpenVMS, HP-UX, TempleOS, z/TPF, XTS-400, Solaris, FreeBSD and MULTICS[95].

Thus, the kernel provides CPU scheduling, file management, memory management, IPC and more (as opposed to microkernel) - as shown in the diagram below. There are a couple of advantages when using this design like: we can use a single static binary for the kernel, we can access all the capabilities of the kernel using system calls and we can get a single execution flow in a single address space[96].

Moreover, probably the biggest disadvantage of a monolithic kernel design is that in case of a service failure the entire system fails. A monolithic kernel can be extended (without recompilation and linking) only if it is modular. Linux it is done using loadable kernel modules, the support for that is defined by enabling “CONFIG_MODULES” which compiling the kernel[97]. Lastly, in kernel version 6.7.4 “CONFIG_MODULES” is referenced in 158 files across the kernel source code files[98].



Monolithic Kernel

[95] https://en.wikipedia.org/wiki/Monolithic_kernel
[96] https://www.geeksforgeeks.org/monolithic-kernel-and-key-differences-from-microkernel/
[97] https://elixir.bootlin.com/linux/latest/source/Makefile#L1106
[98] https://elixir.bootlin.com/linux/latest/A/ident/CONFIG_MODULES

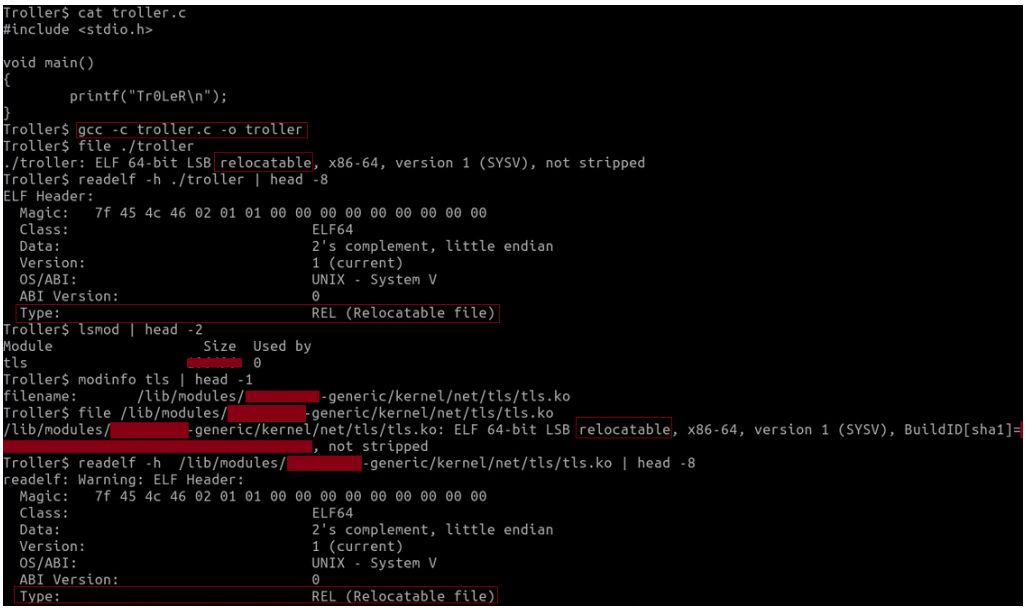
Loadable Kernel Module (LKM)

A loadable kernel module (LKM) allows us to add code to the Linux kernel without the need of recompiling and linking the kernel binary. This is used for different use cases such as (but not limited to) filesystem drivers and device drivers[99].

Overall, when compiling the Linux kernel we can decide if we want to incorporate a specific kernel module as part of the kernel itself (also called built-in kernel modules - more details on them in future writeups) or as a separate "*.ko" (kernel object) file. In case the kernel is already compiled we only have the option of creating a kernel object file which can be later loaded by using the "insmod" utility[100].

Moreover, a kernel object file has the same type as an ordinary object file (before it is linked and can be executed in user mode) - as shown in the screenshot below. This type is called relocatable and is defined as "ET_REL"[101].

Lastly, we can dynamically check the list of the actively loaded kernel modules using the "lsmod"[102] utility - as shown in the screenshot below. "lsmod" basically parses "/proc/modules"[103].



[99] https://tldp.org/HOWTO/Module-HOWTO/x73.html
[100] https://man7.org/linux/man-pages/man8/insmod.8.html
[101] https://man7.org/linux/man-pages/man5/elf.5.html
[102] https://man7.org/linux/man-pages/man8/lsmod.8.html
[103] https://man7.org/linux/man-pages/man5/proc.5.html

Builtin Kernel Modules

In general, we can compile a kernel module[104] as a separate "*.ko" file or include it as part of the kernel itself. If the kernel module is included as part of the kernel's image it is referred to as a builtin kernel module.

Overall, we can check this configuration regarding our compiled kernel by leveraging "/proc/config.gz"[105] or "/boot/config-$(uname-r)"[106] in case they exist. If there are built-in kernel modules compiled into the kernel, it is not sufficient to use "lsmod" for identifying them and we need to use other techniques like searching for specific symbols in "/proc/kallsyms"[107] or using "modinfo"[108] - as shown in the screenshot below.

Lastly, "built-in kernel modules" can't be unloaded as opposed to "loadable kernel modules". However, we can be sure that they are loaded and we don't need to take care of additional files which are needed as in the case of "loadable kernel modules"[109]. Due to that (and more), "built-in kernel modules" are great to ensure essential functionality that must be available at all times[110].

[104] https://medium.com/@boutnaru/the-linux-concept-journey-loadable-kernel-module-lkm-5eaa4db346a1
[105] https://medium.com/@boutnaru/the-linux-concept-journey-proc-config-gz-34c4086e0207
[106] https://medium.com/@boutnaru/the-linux-concept-journey-boot-config-uname-r-6a4dd16048c4
[107] https://www.unix.com/man-page/redhat/8/kallsyms/
[108] https://linux.die.net/man/8/modinfo
[109] https://stackoverflow.com/questions/22929065/difference-between-linux-loadable-and-built-in-modules
[110] https://shape.host/resources/kernel-modules-vs-built-in-making-the-right-choice

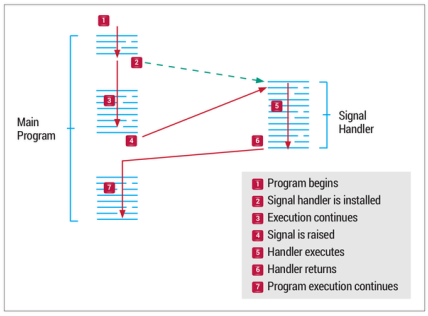
Signals

Signals are asynchronous events sent to a task (process), signals are numbered and their names in Linux start with “SIG” . A couple of examples are: “SIGINT” (generated when Control+C is pressed), “SIGALARM” (generated when the timer set by an alarm is fired), “SIGSTOP” (tells Linux to pause a process execution), “SIGCONT” (tells Linux to resume the execution of a process), “SIGSEGV” (generated in case of a segmentation fault) and “SIGKILL” (when sent to a process causes it to be terminated). When a signal occurs one of three things can happen: the signal is ignored, the signal is caught and handled using a registered function and letting the default action of the signal to happen[111].

Moreover, the default action could be: ignore, terminate, terminate and core dump, stop/pause the process or resume the stop/paused process (they are also called “Signal Dispositions”. Signals can be used as an IPC (Inter Process Communication) mechanism. In case we want to alter the disposition of a signal we can use the “signal”[112] syscall or the “sigcation”[113] syscall - the first is the preferred way.

Overall, for sending a signal by using its PID we can use the “kill”[114] system call. We can also use the “pidfd_send_signal” syscall for sending a signal by using a PID file descriptor[115]. In order to send a signal to a specific thread of a process we can use the “tgkill”[116] syscall. For more syscalls/library calls related to signals I suggest going over the signal man page[117].

Lastly, when a signal is received the relevant signal handler is called - as shown below[118]. The number representing a signal ranges from 1-31, there are also “real time signals” which range from 34-64[119]. For further information I suggest going over the relevant source code as part of the Linux kernel[120].



[111] https://faculty.cs.niu.edu/~hutchins/csci480/signals.htm
[112] https://man7.org/linux/man-pages/man2/signal.2.html
[113] https://man7.org/linux/man-pages/man2/sigaction.2.html
[114] https://man7.org/linux/man-pages/man2/kill.2.html
[115] https://man7.org/linux/man-pages/man2/pidfd_send_signal.2.html
[116] https://man7.org/linux/man-pages/man2/tgkill.2.html
[117] https://man7.org/linux/man-pages/man7/signal.7.html
[118] https://devopedia.org/linux-signals
[119] https://www-uxsup.csx.cam.ac.uk/courses/moved.Building/signals.pdf
[120] https://elixir.bootlin.com/linux/v6.7/source/kernel/signal.c

Real Time Signals

In general, “Real Time Signals” (aka realtime signals or rtsignals) are similar to normal/original signals[121] in the sense they have signal numbers and can be dealt with using the original signal functions. Since kernel 2.2, Linux started supporting real-time signals; their range is defined using the SIGRTMIN and SIGRTMAX macros. Linux supports 33-different real-time signals (ranging from 32-64). However, the range of available real-time signals varies according to the glibc threading implementation (this variation can occur at run time according to the available kernel and glibc), due to that programs should never refer them using hard-coded numbers[122].

Overall, there are some differences between the original signals and real-time signals. Among them is the fact that real-time signals have multiple instances of individual real-time signals that can be queued (as opposed to the original signals which are merged when sent to a process that has a signal pending). Also, real-time signals can carry additional data with the signal number[123].

Moreover, regarding the delivery priority lower number signals are given priority over higher number signals. For Linux this is true across the types of all signals, due to that the original signals have higher priority than all real-time signals. By the way, POSIX says that priority of real-time versus ordinary signals is unspecified, and only the real-time signals have a specified priority relative to each other[124].

Lastly, for handling real-time signals and receiving the additional data we need to use the “sigaction” system call[125] or library call[126] and not “signal” system call[127]. For sending a signal we should use the “sigqueue” syscall[128] or the “sigqueue” library call[129] and not the “kill” syscall[130] or “the kill” library[131] - as shown below[132].

```
#include <signal.h>

union sigval {
    int   sival_int;
    void *sival_ptr;
};

int sigqueue (pid_t pid, int sig, const union sigval value);
```

[121] https://medium.com/@boutnaru/the-linux-concept-journey-signals-d1f37a9d2854
[122] https://man7.org/linux/man-pages/man7/signal.7.html
[123] https://www.nxp.com/docs/en/white-paper/CWLNXRTOSWP.pdf
[124] https://davmac.org/davpage/linux/rtsignals.html
[125] https://linux.die.net/man/2/sigaction
[126] https://linux.die.net/man/3/sigaction
[127] https://man7.org/linux/man-pages/man2/signal.2.html
[128] https://linux.die.net/man/2/sigqueue
[129] https://linux.die.net/man/3/sigqueue
[130] https://linux.die.net/man/2/kill
[131] https://linux.die.net/man/3/kill
[132] https://www.softprayog.in/programming/posix-real-time-signals-in-linux

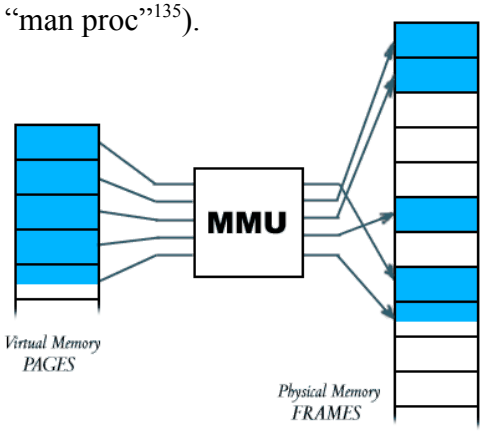
Memory Management - Introduction

Linux's memory management subsystem is surprisingly responsible for managing the system held in the system. It does that by using several components: virtual memory and demand paging, userspace memory allocator and an allocator for kernel structures.

Virtual memory is a memory management technique used to create an illusion for a running program that it has the entire memory address space only for itself with no need to coordinate the address used with other running programs. Two basic concepts we should get to know are frames (sometimes called physical pages/page frames) and pages (sometimes called virtual pages).

First, we are going to divide physical memory (total available memory the system has, we can see it using "cat /proc/meminfo | grep -i memtotal") into blocks with the same size. We call them "frames", their size is a power of 2 (for now we are going to use the magic number of 4K more on that in future writeup). Second, we divide the virtual memory address space into blocks with the same size called "pages". The maximum range of the address space depends on the CPU and the OS (more on that in a future writeup). The size of the "pages" is also a power of 2, for simplicity the size of "pages" equals those of "frames". We can see that size from AUXV[133].

For those who know NAT/PAT (Network Address Translation/Port Address Translation) it is similar to the process done by the memory management subsystem. We take a "Virtual Address" which resides in a specific "page" and it is translated into a "Physical Address" which resides in a specific "frame". The component responsible for that is the MMU (Memory Management Unit), for now we are going to talk about hardware based only MMUs (we will cover soft-MMU in the future). The translation is based on tables created and managed by the OS ("page tables") and the MMU is responsible for using them for the translation and the enforcement of different checks (defined by page bit-like validity) - see the illustration below[134]. In order to configure the memory management subsystem we can use the "/proc/sys/vm" interface (for more information about that I suggest reading "man proc"[135]).



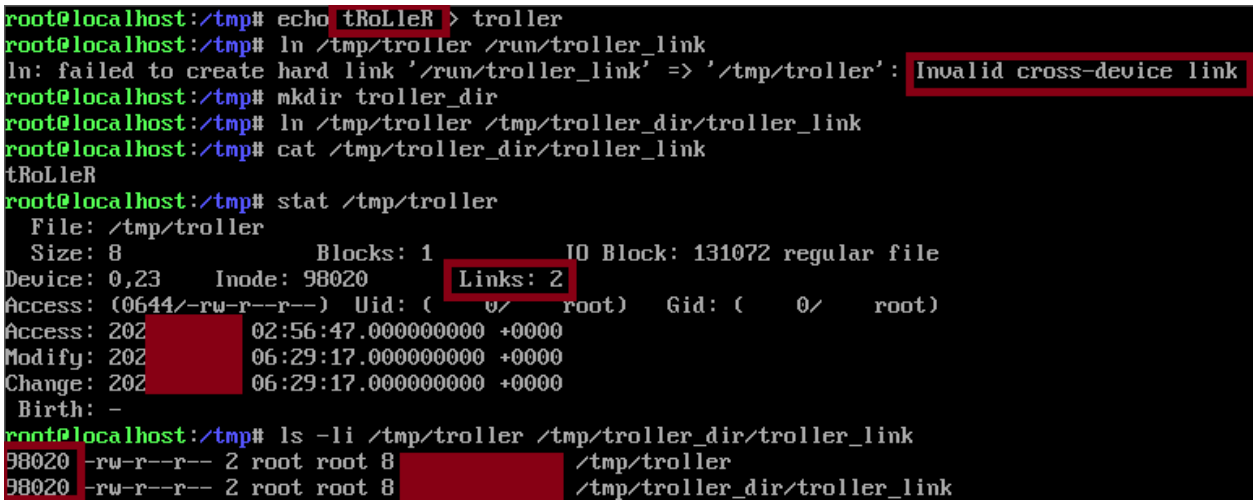
[133] https://medium.com/@boutnaru/linux-the-auxiliary-vector-auxv-cba527871b50
[134] http://dysphoria.net/OperatingSystems1/4_paging.html
[135] https://man7.org/linux/man-pages/man5/proc.5.html

Hard Link

As mentioned in previous writeup, an inode is data structure used by Unix/Linux like filesystems in order to describe a filesystem object[136]. Thus, each hard link has the same inode value as the original file it points to - as shown in the screenshot below. When removing a hard link it just reduces the "link count" (think about it as a reference count), but it does not affect other links (the file is removed only when the count reaches "0"). Each hard link has a different file name and if the size of the content of one link changes then all the hard links file sizes are updated[137].

Overall, we have a couple of limits regarding hard links which includes the fact we can create a hard link only for regular files (not including special files or directories). Also, we can not use a hard link to point to a file in a different filesystem[138] - as shown in the screenshot below.

Lastly, we can use the "ln" command line utility in order to create hard links[139] - as shown in the screenshot below. It is based on the "link"/"linkat" system call which is described as "make a new name for a file"[140].



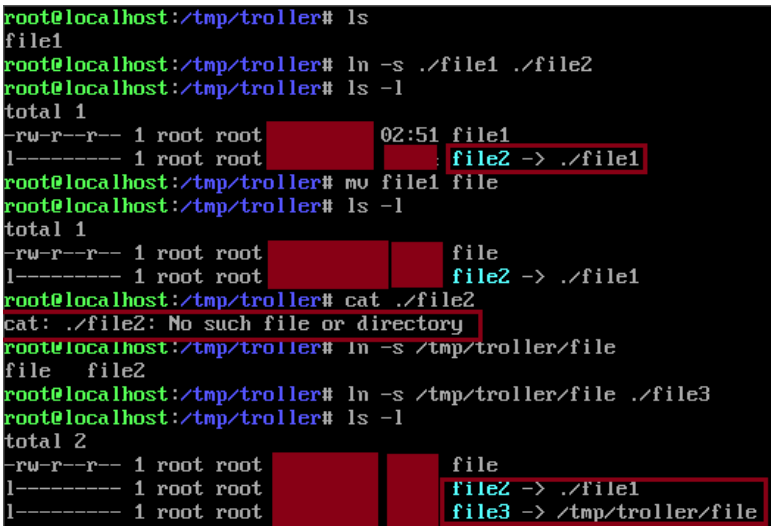
[136] https://medium.com/@boutnaru/linux-what-is-an-inode-7ba47a519940
[137] https://www.geeksforgeeks.org/soft-hard-links-unixlinux/
[138] https://www.redhat.com/sysadmin/linking-linux-explained
[139] https://man7.org/linux/man-pages/man1/ln.1.html
[140] https://man7.org/linux/man-pages/man2/link.2.html

Soft Link

As opposed to a hard link[141] which points to an inode, a soft link (aka symbolic link) points to another directory/file by name. Thus, soft links do not have the limitation of hard links in the sense of not being able to point to directories[142]. By the way, we can use the "ln" command with the "-s" in order to create a soft link[143] - as shown in the screenshot below.

Moreover, we can use soft links to point to a file/directory in a different file system, having a different inode, different permissions and size. Because a soft link is not a mirror (as in the case of a hard link) the size of the file are the number of bytes needed to hold the name of the file/directory[144]. The name of the file/directory can be based on a relative/full path - as shown in the screenshot below.

Lastly, soft links have their own limitations (we can't have only pros ;-). In case the name of the original file/directory (the target of the soft link) is changed it "breaks" the pointer/link - as shown in the screenshot below. Also, changing the permissions of the original file/directory does not affect the soft link. The creation of a soft link is based on the "link"/"linkat" system call which is described as "make a new name for a file"[145].



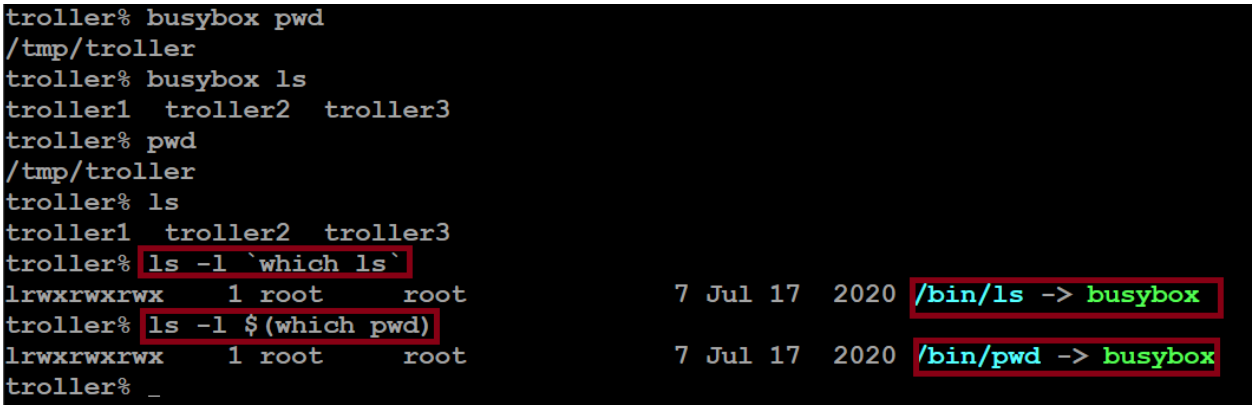
[141] https://medium.com/@boutnaru/the-linux-concept-journey-hard-link-f3e9b3d6b8c4
[142] https://kodekloud.com/blog/linux-create-and-manage-soft-links/
[143] https://man7.org/linux/man-pages/man1/ln.1.html
[144] https://www.redhat.com/sysadmin/soft-links-linux
[145] https://man7.org/linux/man-pages/man2/link.2.html

BusyBox

BusyBox sees itself as the “The Swiss Army Knife of Embedded Linux”. It is a software component that combines tiny versions of many Unix utilities into a single binary. By doing so it can replace most utilities like GNU shellutils and fileutils. It is important to know that BusyBox has fewer options than the full-featured GNU utilities[146].

Moreover, BusyBox supports about 400 commands such as: “ls”, “ln”, “mv”, “mkdir”, “more” and “grep”. Also, it is an open source (GPL) project[147]. We can execute any command supported by BusyBox in the following way: “busybox <command>” - as shown in the screenshot below. The screenshot was taken from a “Buildroot Linux”[148]. We can download the source code of BusyBox for the official BusyBox website[149]. There are also porting of BusyBox to other operating systems like Android[150].

Lastly, we can also execute BusyBox commands without calling the busybox executable directly. This works due to the fact that for every command supported by BusyBox a symbolic link with the name of the command is created. The symbolic link points to the BusyBox executable - as shown in the screenshot below. This causes the “cmdline”, which holds the complete command line for the process (unless it is a zombie process) to hold the name of the symbolic link and the arguments passed[151].



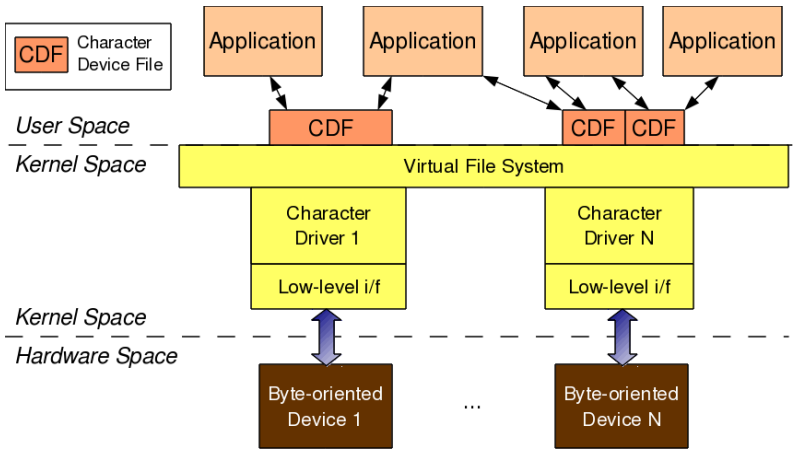
[146] https://busybox.net/about.html
[147] https://opensource.com/article/21/8/what-busybox
[148] https://copy.sh/v86/?profile=buildroot
[149] https://busybox.net/downloads/
[150] https://github.com/meefik/busybox
[151] https://man7.org/linux/man-pages/man5/proc.5.html

Character Devices

In Unix\Linux hardware devices are accessed using device files (located in "/dev"). A character device is used in case of slow devices (like sound card/joystick/keyboard/serial ports), which usually manage a small amount of data. Operations on those devices are performed sequentially byte by byte[152].

Moreover, as with every device also character devices have a major number and a minor number[153]. We can think of the major number identifying the driver and the minor number identifying each physical device handled by the driver. Thus, we can say we have four main entities: the application, a character device file, a character device driver and a character device - as shown in the diagram below[154].

Lastly, in order to add a character device driver we need to register it with the kernel. This can be done by leveraging the "register_chrdev" function which is part of the "include/linux/fs.h" header file[155]. In the case of kernel version "6.9.7" there are 46 files which reference that function[156].



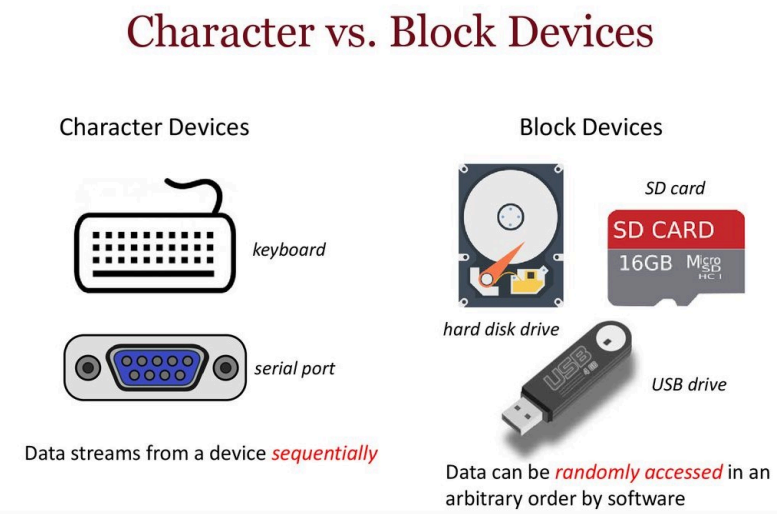
[152] https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html
[153] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[154] https://sysplay.in/blog/linux-device-drivers/2013/05/linux-character-drivers/
[155] https://elixir.bootlin.com/linux/v6.9.7/source/include/linux/fs.h#L2746
[156] https://elixir.bootlin.com/linux/v6.9.7/A/ident/register_chrdev

Block Devices

Block devices provide the ability to randomly access data which is organized in fixed-size blocks. Examples of such devices are: RAM disks, CD-ROM drives and hard drives. The speed of block devices is in general higher than those of character devices[157]. Another difference is that a character device has a single current position. However, in the case of a block device we need to be able to move to any random position for accessing/writing data[158]. We can see a comparison (with examples) between character devices and block devices in the diagram below[159].

Moreover, as with every device, block devices have a major number and a minor number[160]. We can think of the major number identifying the driver and the minor number identifying each physical device handled by the driver. Although there is a difference between block devices they have some common abstractions like: data is typically buffered/cached on reads (an even writes if supported) and data if mostly organized as files and directories for ease of use by the user[161].

Lastly, in order to add a character device driver we need to register it with the kernel. This can be done by leveraging the "register_blkdev" macro which is part of the "include/linux/fs.h" header file[162]. In the case of kernel version "6.9.7" there are 33 files which reference that macro[163].



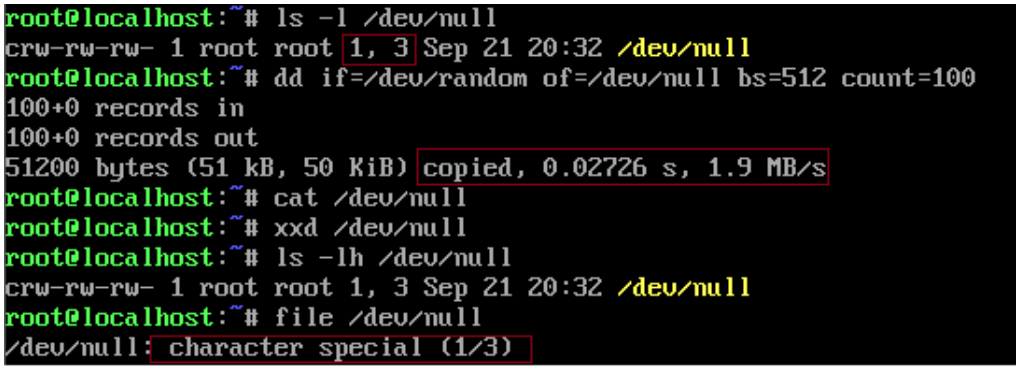
[157] https://medium.com/@boutnaru/the-linux-concept-journey-character-devices-0c75aa70ceb2
[158] https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html
[159] https://www.csltraining.com/block-device-vs-character-devices-in-linux-os/
[160] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[161] https://www.codingame.com/playgrounds/2135/linux-filesystems-101---block-devices/about-block-devices
[162] https://elixir.bootlin.com/linux/v6.9.7/source/include/linux/blkdev.h#L809
[163] https://elixir.bootlin.com/linux/v6.9.7/A/ident/register_blkdev

Null Device (/dev/null)

The “Null Device” (/dev/null) is a character device[164] used under Linux. This device ignores all data written to but returns that the write operation has succeeded[165] - as shown in the screenshot below (taken using https://copy.sh/v86/?profile=archlinux). Due to its nature the null device it has a special “lseek” function called “null_lseek” so we can use “fopen” on it with an append flag[166].

Overall, the null device has its own “file operations” struct aka “null_fops”[167]. From that structure we can find the read function of the device “read_null” which returns “0” for all input[168]. By default this device has a <1,3> as its <major,minor> number pair[169] - as also shown in the screenshot below.

Lastly, the null device is not unique only for Unix/Linux operating systems. We can find it on Windows as “\Device\Null”, on DOS or CP/M as “\DEV\NUL”, on OS/2 as “nul”, on OpenVMS as “NL:” and more. Due to its nature “/dev/null” is used for disposing of unwanted output streams of a process or as an empty file for input streams. Because it is not a directory we can use the “mv” utility with it[170].



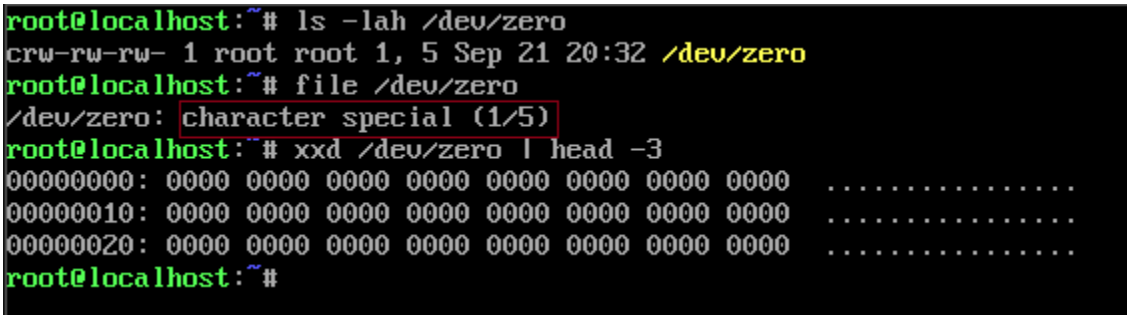
[164] https://medium.com/@boutnaru/the-linux-concept-journey-character-devices-0c75aa70ceb2
[165] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L436
[166] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L566
[167] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L649
[168] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L430
[169] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[170] https://en.wikipedia.org/wiki/Null_device

Zero Device (/dev/zero)

The “Zero Device” (/dev/zero) is a character device[171] used under Linux. The goal of the device is to provide null characters (ASCII code of 0x00) - as shown in the screenshot below (taken using https://copy.sh/v86/?profile=archlinux). Due to its nature the zero device it has a special “lseek” function called “null_lseek” so we can use “fopen” on it with an append flag[172]. This is the same function for lseek as with with the “Null Device”[173].

Overall, the zero device has its own “file operations” struct aka “zero_fops”[174]. From that structure we can find the “read_iter” function of the device “read_iter_zero” which is responsible for returning zeros[175]. By default this device has a <1,5> as its <major,minor> number pair[176] - as also shown in the screenshot below.

Lastly, as opposed to “/dev/null”[177] we can use “/dev/zero” both as a source or destination in different commands/utilities (while “/dev/null” can be only used as a sink for data). In case we memory map “/dev/zero” it is like using anonymous memory[178]. This can be done by leveraging the “mmap” syscall[179] or the “mmap” library function[180].



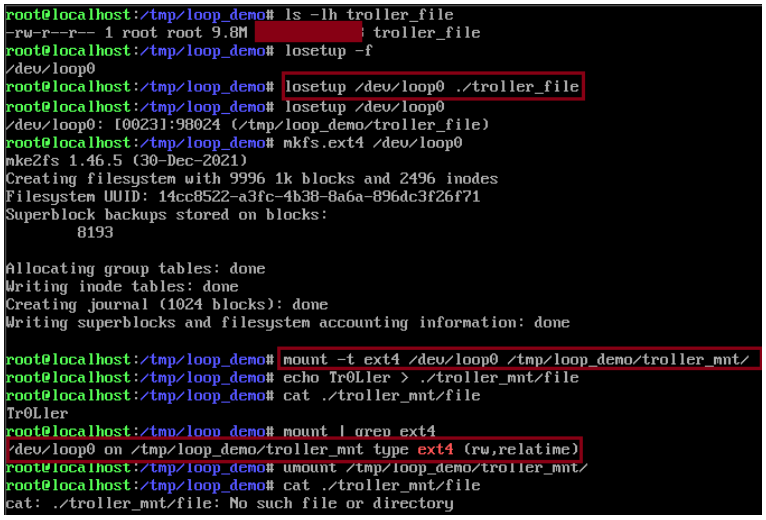
[171] https://medium.com/@boutnaru/the-linux-concept-journey-character-devices-0c75aa70ceb2
[172] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L566
[173] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L629
[174] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L668
[175] https://elixir.bootlin.com/linux/v6.12/source/drivers/char/mem.c#L471
[176] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[177] https://medium.com/@boutnaru/the-linux-concept-journey-null-device-dev-null-b63a3c42fdb2
[178] https://en.wikipedia.org/wiki//dev/zero
[179] https://man7.org/linux/man-pages/man2/mmap.2.html
[180] https://linux.die.net/man/3/mmap

Loop Device

Under Linux (and other Unix based OSes) we can use a regular file as a block device. Thus, a loop device (sometimes can also be called loopback although not a networking device) is a virtual/pseudo device that allows us accessing the data in a regular file as a block device. The device node of the pseudo-device is available under "/dev". Because of that we can create a new file system and mount it as with an ordinary block device[181]. In other Unix based OSes it called in different name like vnd (vnode disk) on NetBSD/OpenBSD or lofi (loop file interface) on Solaris/OpenSolaris[182].

Overall, after associating a file with a loop device the OS treats that file as if it was a physical disk. By default, loop devices are not persistent across reboots. We can add entries to specific configuration files to make them persistent (think for example about "/etc/fstab"). Another benefit is the ability to dynamically resize a loop device[183].

Lastly, in order to associate loop devices with a regular file (and other operations like detaching, resizing and more) we can leverage the "losetup" command line utility[184] - as shown in the screenshot below taken using "Arch Linux" from copy.sh[185]. For implementation details I suggest going over the source code as part of the Linux kernel[186].



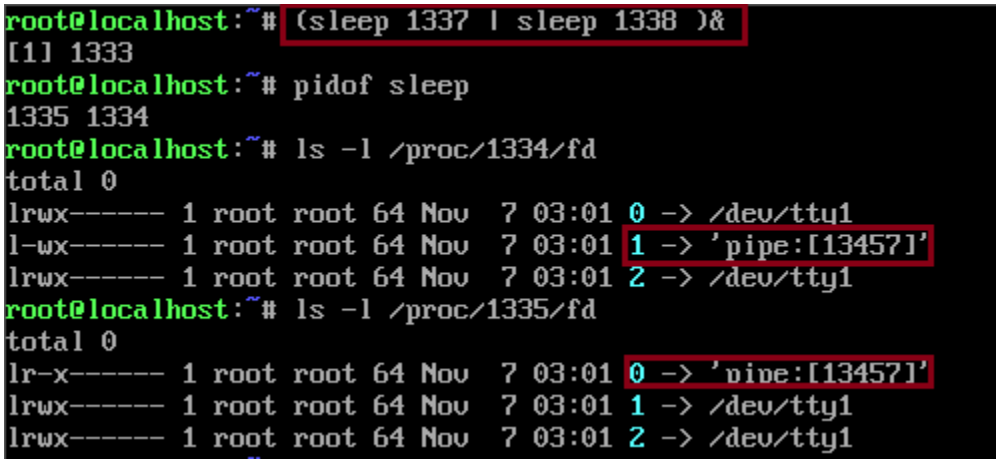
[181] https://dzone.com/articles/loop-device-in-linux
[182] https://en.wikipedia.org/wiki/Loop_device
[183] https://www.lenovo.com/us/en/glossary/loop-device/
[184] https://man7.org/linux/man-pages/man8/losetup.8.html
[185] https://copy.sh/v86/?profile=archlinux
[186] https://elixir.bootlin.com/linux/v6.11.5/source/drivers/block/loop.c

Unnamed Pipe (Anonymous Pipe)

In general a Linux pipe allows commands to send output of one program to a different one. Thus, the term "Piping" means redirecting standard input/output/error of one process to another. In order to create a pipe (unnamed pipe) we can use the "|" character. For example "Command-(i)" | "Command-(i+1)....|..|"Command-(i+n)" - as shown in the screenshot below[187].

Overall, the "unnamed pipe" that is created (like by the command shown above) can't be accessed from a different session. Those types of pipes are created temporarily and deleted after the execution of the command. The pipe can be created using the pipe/pipe system call[188].

Lastly, by using an unnamed pipe we get a unidirectionality of the stream. Also, each shell has its own way of getting the status of each command in case of a pipeline, because by default only the status of the last command is saved to "$?". For example "bash" has the "$PIPESTATUS" environment variable[189].



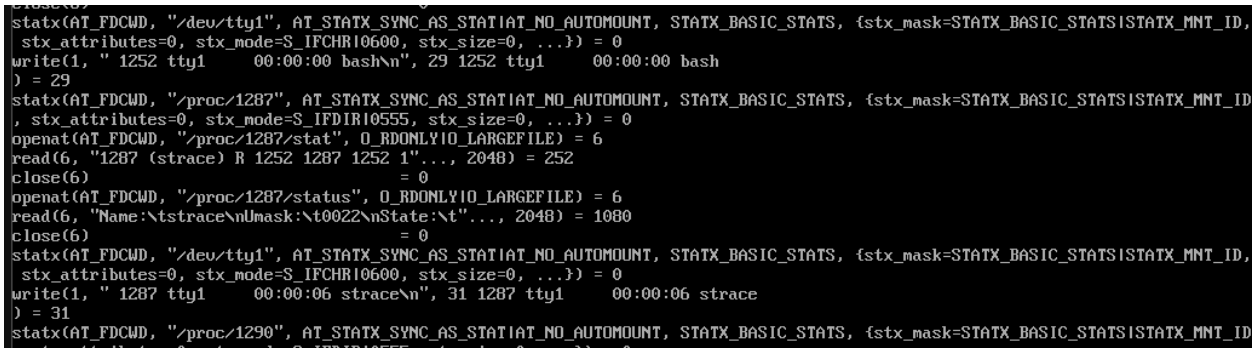
[187] https://opensource.com/article/18/8/introduction-pipes-linux
[188] https://man7.org/linux/man-pages/man2/pipe.2.html
[189] https://www.baeldung.com/linux/anonymous-named-pipes

Processes & Threads

In general, a process is a holder for a running instance of a program, that its executable code (and could be also the shared libraries) loaded from some storage medium. A program can have multiple running instances.If we think about it, a process is a resource management unit for things like: memory address space, open files, threads (at least one), data section, timers, signal handlers, sockets and more. Threads represent the execution flows of a running program (that is also what the kernel schedules). Moreover, threads also include: the program counter, stack and processor registers state[190].

From the Linux perspective threads and processes are the same, you can think of threads as processes sharing the same memory space (more on that in the future). The theory of operating systems states that all the information of a process is managed by a PCB (Process Control Block)—in Linux it is “task_struct”[191]. In parallel there is a TCB (Thread Control Block) that holds the relevant information about threads, under Linux it is also “task_struct”.

In order to access the information about a process under Linux from user-space we can use “/proc” (“man 5 proc”) - as shown in the output of strace below. It is what the command “ps” does (“man ps”) as shown in the screenshot below (from copy.sh while calling “strace `which ps -ef`”). Lastly, under Linux we have to use two different syscalls “fork” (“man 2 fork”) and another one from the “execve” (“man 2 execve”) family in order to create a new process[192].



[190] https://linux-kernel-labs.github.io/refs/heads/master/lectures/processes.html
[191] https://elixir.bootlin.com/linux/v6.1/source/include/linux/sched.h
[192] https://www.softprayog.in/programming/creating-processes-with-fork-and-exec-in-linux

Why never trust only the source code? And verify the created binary (Compiler Optimizations)

One might think if in the source code we see a specific call to a C library function the ELF binary will contain a call to that specific function. Let's take an example (while using gcc), in our source code we call “printf” - as shown in the screenshot below. Using the command “gcc troller.c -o ./troller” we compile and link the source to an ELF binary and execute it. However, using “nm -D” (which lists the dynamic symbols used by the binary[193]) we don’t see “printf” we just see “puts”. We can also verify it using objdump (which disassembles the binary[194]) - as shown in the screenshot below.

Moreover, the only reason that optimization was possible is because we have not passed a formater to “printf” and that we added a newline at the end of the string. Without the newline the optimizer could not use “puts” because it adds a trailing newline[195].

Lastly, those optimizations can also lead to security vulnerabilities (more on that in future writeups). Also, the reason we see in objdump’s output “puts@plt” and not just “puts” is due to the way dynamic symbols are located/used in the case of dynamic linking.

```
Troller # cat ./troller.c
#include <stdio.h>

void main()
{

        printf("Hey !!! Troller\n");


}
Troller # gcc troller.c -o ./troller
Troller # ./troller
Hey !!! Troller
Troller # nm -D ./troller
                 w __cxa_finalize@GLIBC_2.2.5
                 w __gmon_start__
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 U __libc_start_main@GLIBC_2.34
                 U puts@GLIBC_2.2.5
Troller # objdump -D ./troller | grep printf
Troller # objdump -D ./troller | grep puts
0000000000001050 <puts@plt>:
    1054:       f2 ff 25 75 2f 00 00    bnd jmp *0x2f75(%rip)        # 3fd0 <puts@GLIBC_2.2.5>
    115b:       e8 f0 fe ff ff          call   1050 <puts@plt>
```

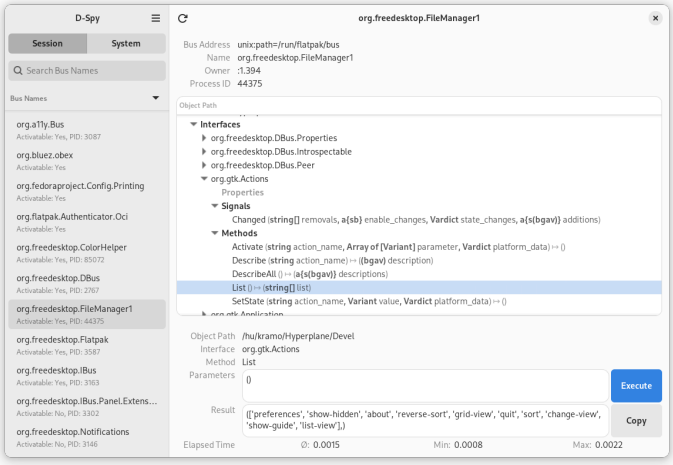
[193] https://linux.die.net/man/1/nm
[194] https://linux.die.net/man/1/objdump
[195] https://linux.die.net/man/3/puts

D-BUS (Desktop Bus)

D-BUS (Desktop BUS) is a message bus system (based on the dbus protocol) which allows applications to pass information between each other. Beside providing IPC (inter-process communication) can help in coordinating the lifecycle of processes. D-BUS is mostly implemented using a dbus daemon (which can be executed system wide/per user session) and a set of libraries[196]. There are different implementations for D-BUS such as: “GDBus”, “sd-bus” and “kdbus”. Probably the most widely used is the reference implementation “libdbus”[197], developed by the same freedesktop.org project that designed the specification[198].

Overall, D-BUS provides both IPC and RPC (Remote Procedure Call) mechanisms. Objects are uniquely identified through the combination of a bus name and object path[199]. We can use different debugging tools for exploring D-BUS connections such as: “busctl”[200], “Qt D-BUS Viewer”[201] or “D-Spy”[202] - as shown in the screenshot below.

Lastly, examples (but not limited to) for projects using\leveraging D-BUS are: “KPlayer”, “Ark”, “Pidgin”, “Gossip”, and “Avahi”[203]. All the D-BUS discussion can be found as part of the dbus mailing list[204]. Also, there could be multiple buses for example Linux desktops are using two: “System Bus” (relevant for all users/processes) and “Session Bus” - as shown in the GUI screenshot below. Based on permissions a process can connect to every bus.



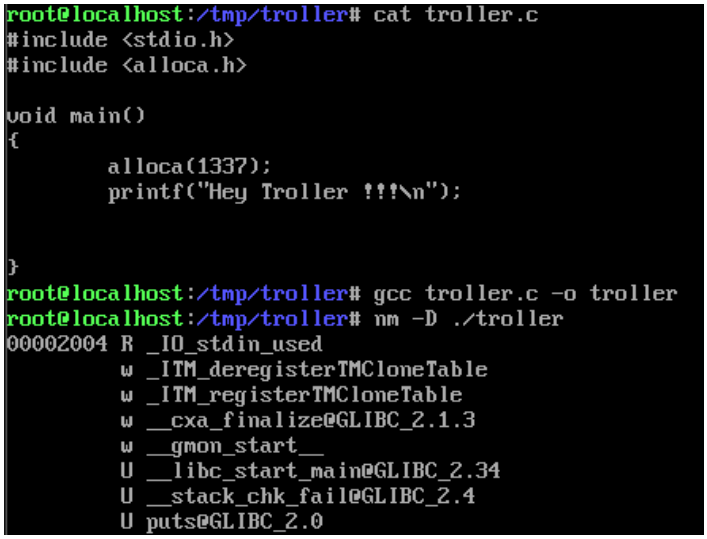
196 https://www.freedesktop.org/wiki/Software/dbus/
197 https://gitlab.freedesktop.org/dbus/dbus
198 https://en.wikipedia.org/wiki/D-Bus
199 https://pythonhosted.org/txdbus/dbus_overview.html
200 https://man.archlinux.org/man/busctl.1
201 https://doc.qt.io/qt-6/qdbusviewer.html
202 https://apps.gnome.org/Dspy/
203 https://www.freedesktop.org/wiki/Software/DbusProjects/
204 https://lists.freedesktop.org/mailman/listinfo/dbus/

The Weirdness Behind the Implementation of “alloca”

As part of the C programming language we have the library function called “alloca”. It is used in order to allocate bytes of space on the stack frame of the caller. Thus, this space is automatically freed when the function called “alloca” returns. Due to that it is also faster than malloc and free[205].

Overall, it is important to understand that after building the binary there won’t be any call to “alloca” - as shown in the screenshot below. The reason for that is if there was a function call to “alloca” a new stack frame would have been created and the calling function wouldn't be able to access the data on the stack. Due to that, we can’t change its behavior by linking to another implementation.By the way, those who are wondering why there is a symbol of “puts” and not “printf” although in our code calls “printf” I suggest reading my previous writeup about it[206].

Lastly, the implementation of “alloca” is machine/compiler dependent. For example gcc replaces it with “__builtin_alloca()”. It does not validate the parameters of the function so it can exceed the limit of the stack if the caller does not check for it[207].



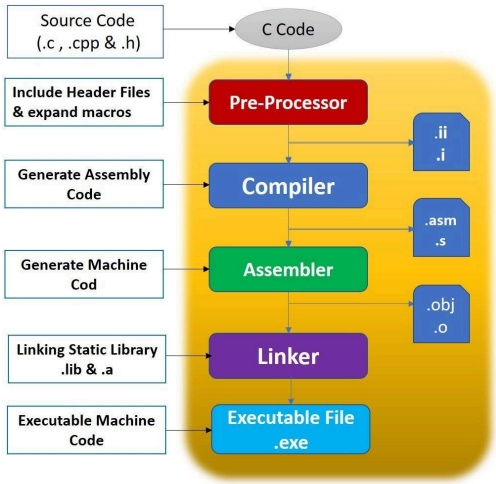
[205] https://man7.org/linux/man-pages/man3/alloca.3.html
[206] https://medium.com/@boutnaru/code-optimizations-why-never-trust-only-the-source-code-and-verify-the-created-binary-c24047427251
[207] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

GNU Toolchain

A toolchain is a set of programming utilities/tools used for building/running and even debugging applications/OSes[208] - as shown in the flow diagram below[209]. Among those tools are: compilers, debuggers, assemblers, linkers and more[210].

Overall, in the case of the GNU toolchain it is used for building the Linux kernel and other Linux applications. This toolchain includes: “GNU make”, “GNU Compiler Collections” (gcc), “GNU Debugger” (gdb), “GNU Build System” (autotools), “GNU C Library” (glibc or eglibc) and “GNU Binutils” (linker, assembler and other library/object manipulation tools)[211].

Lastly, the GNU toolchain is available for different processor architectures like: ARM, x86-64, SupserH, RISC-V, MIPS, PowerPC, IA-32, IA-64, PA-RISC, HC12 and more[212]. By the way, they are also portings of the GNU toolchain for Windows[213].



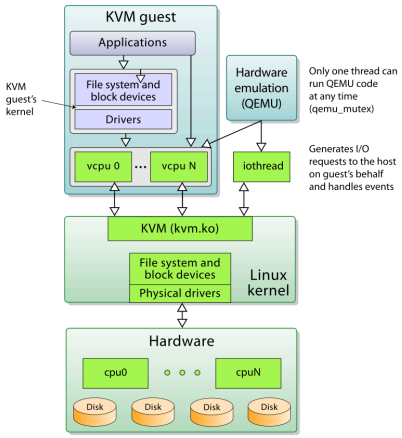
[208] https://www.jetbrains.com/help/clion/how-to-create-toolchain-in-clion.html
[209] https://microcontrollerslab.com/embedded-systems-build-process-using-gnu-toolchain/
[210] https://stackoverflow.com/questions/50104079/what-exactly-is-a-toolchain
[211] https://developer.arm.com/documentation/den0013/d/Tools--Operating-Systems-and-Boards/Software-toolchains-for-ARM-pro cessors/GNU-toolchain
[212] https://en.wikipedia.org/wiki/GNU_Compiler_Collection
[213] https://gnutoolchains.com/

KVM (Kernel-based Virtual Machine)

KVM (Kernel-based Virtual Machine) is an open source full virtualization solution for Linux. It is targeting the x86 architecture. It leverages the "Intel VT" or "AMD-V" virtualization extensions. The main components of KVM are: "kvm.ko" (provides the core virtualization infrastructure). By using KVM we can execute multiple VMs running unmodified versions of OSes (like Windows and Linux). Every VM can have its own virtual: CPUs, memory and more[214]. It is important to know that KVM emulates only a subset of hardware devices. Thus, KVM defers to other client applications such "FireCracker", "crosvm" and "QEMU" for device emulation - as shown in the diagram below[215].

Overall, KVM was first introduced in October 2006 by "Avi Kivity" using a post in the "Linux Kernel Mailing List". As part of his post he describes that patchset adds a driver for Intel's virtualization extension which is exposed using a chartered device ("/dev/kvm"). This is done to enable the usage of virtualization capabilities from userspace[216]. After the support for Intel's VMX instructions the support for AMD's SVM instructions was introduced in November 2006[217]. By the way, KVM has been ported to other operating systems (beside Linux) like FreeBSD[218].

Lastly, the KVM patch has been part of the kernel since version 2.6.20 (February 2007). Today they are different projects which leverage KVM as their default hypervisor. Probably the most known are OpenStack and oVirt[219]. For reference I suggest going over the code of KVM as part of the Linux kernel[220]. Although KVM had been originally designed for x86 processors, support for other processors (like ARMand PowerPC) were added[221].



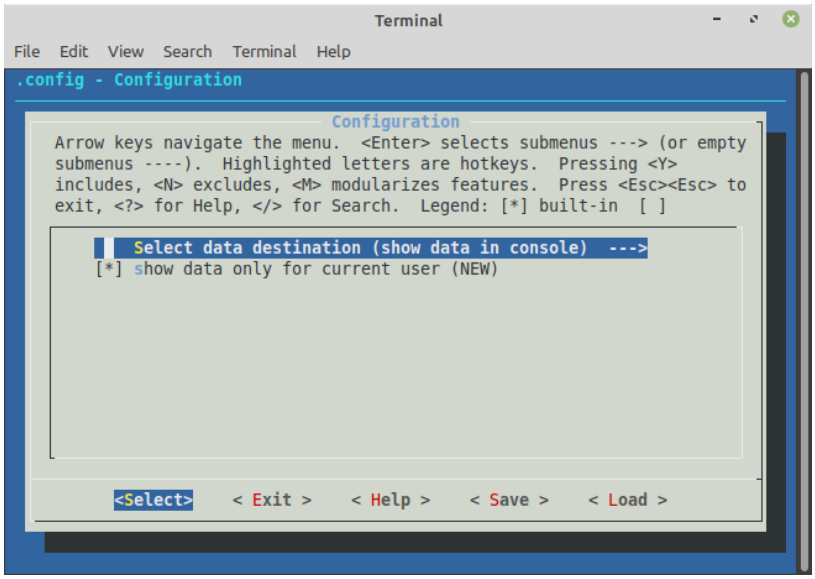
[214] https://linux-kvm.org/page/Main_Page
[215] https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine
[216] https://lkml.iu.edu/hypermail/linux/kernel/0610.2/1369.html
[217] https://lkml.iu.edu/hypermail/linux/kernel/0611.3/0850.html
[218] https://docs.freebsd.org/en/books/handbook/virtualization/
[219] https://lwn.net/Articles/705160/
[220] https://elixir.bootlin.com/linux/v6.12.1/source/virt/kvm
[221] https://www.linux-kvm.org/page/Processor_support

Kconfig

Kconfig is a selection based configuration system. It was originally developed for the Linux kernel (but can and used by other projects today). We can use it in order to select build time options and/or enable\disable features[222]. With Kconfig there are different config interpreters that can be used like: “config”, “nconfig” (menu based), “menuconfig” (also menu based), “xconfig” (QT based), “gconfig” (GTK+ based) and more[223].

Overall, all the configuration parameters are defined in a Kconfig format (more on the format in a future writeup). Different Kconfig interpreters \utilities parses the config file and displays (CLI\menu\GUI) the different configuration with the ability to modify it - as shown in the screenshot below[224].

Lastly, the configuration options are also called “symbols”. We can define dependencies between symbols (as part of the Kconfig files) to enforce what configuration is valid. Also, symbols can be grouped into menus/submenus to organize the config interface[225]. For more information I suggest also going over the Kconfig documentation as part of the Linux kernel documentation[226].



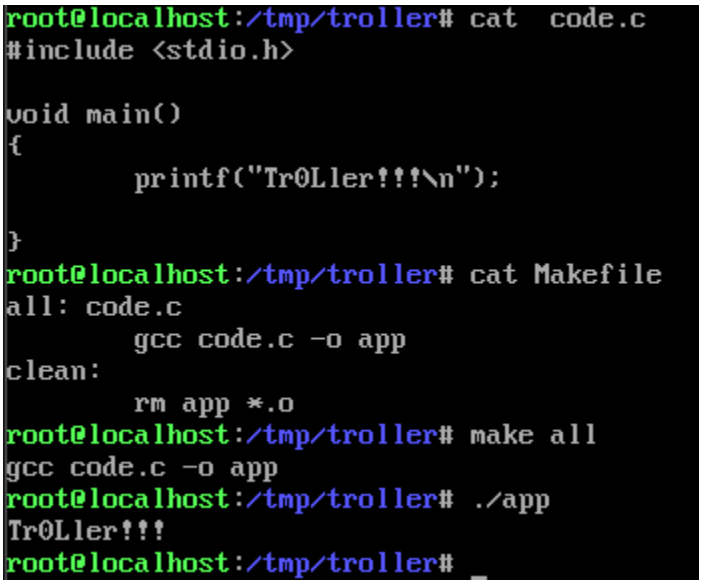
[222] https://docs.legato.io/21_05/toolsKconfig.html
[223] https://devpress.csdn.net/linux/62f636b9c6770329307fc366.html
[224] https://habr.com/en/articles/515398/
[225] https://docs.zephyrproject.org/latest/build/kconfig/index.html
[226] https://docs.kernel.org/kbuild/kconfig-language.html

Makefile

The goal of using “Makefiles” is for helping us in deciding the portion of a large program we want to compile/recompile. We can also use “Makefiles” to run a series of instructions based on what files have been changed. Makefiles are very popular for C\C++ development under Linux, however there are also alternative build systems such as: CMake, Bazel and Ninja. A “Makefile” is composed by a set of rules, which includes targets (files names separated by spaces), prerequisites (the file names which need exist before running the commands of the targets) and commands which have to start with tabs for indentation[227] - as shown in the screenshot below.

Overall, we can think about makefiles as a declarative programming language with conditions but without describing the order of execution between targets. It is important to know that we can define macros in makefiles which act like variables and can be overridden using command line arguments passed to the Make utility[228]. Also we can have include statements, pattern matching rules and comments which are marked using “#”[229].

Lasty, the make utility is used to parse Makefiles and thus automates the building process of an application from source code[230]. For a reference Makefile I suggest going over the main Makefile of the Linux kernel[231].



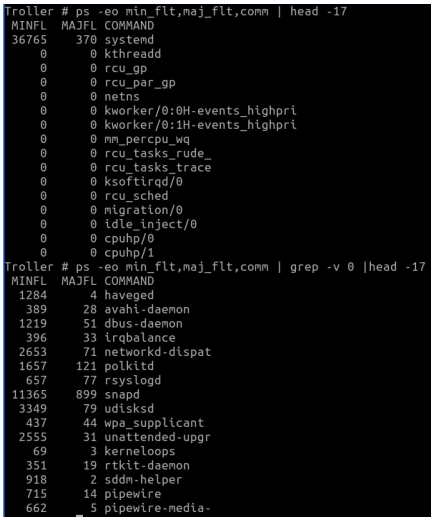
[227] https://makefiletutorial.com/
[228] https://linux.die.net/man/1/make
[229] https://en.wikipedia.org/wiki/Make_(software)
[230] https://www.redswitches.com/blog/make-command-in-linux/
[231] https://github.com/torvalds/linux/blob/master/Makefile

Page Faults

After understanding “Demand Paging”[232] it is time to deep dive into “Page Faults”. We can split page faults into two types: “Major Page Faults” and “Minor Page Faults”. But before we talk about them let us define what is a “Page Miss” and a “Page Hit”.“Page Hit” describes the case in which the CPU references an address in a page and that page is loaded into memory. On the other end, “Page Miss” is the opposite case in which the page we want to reference is not loaded into memory.

Basically, a “Major Page Fault” is the case where the page we want to reference exists only on the system disk, thus reading it will be an expensive operation. This type is also called “Hard Page Fault”. On the other hand, “Minor Page Fault” is a case in which the required page is not stored on disk but in some other place in memory. An example for that is in the working set of another process. It is also sometimes called “Soft Page Fault”.

The Linux kernel holds statistics about both minor and major page faults. They are saved as part of “task_struct” (current->maj_flt and and current->min_flt)[233]. The relevant function that handles the accounting is “mm_account_fault”[234]. It is crucial to understand that kernel threads never page fault or should access user space addresses. The only exception for that is if vmalloc() is used. If kmalloc() is used the pages are never paged. More about vmalloc() and kmalloc() in the future. In order to demonstrate what we have learned we can execute the following command “ps -eo min_flt,maj_flt,comm” that will show us the statistics for major and minor page faults on our system. You can see the output in the screenshot below, we can see that for each kernel thread the values are zero and for other processes we have different numbers.



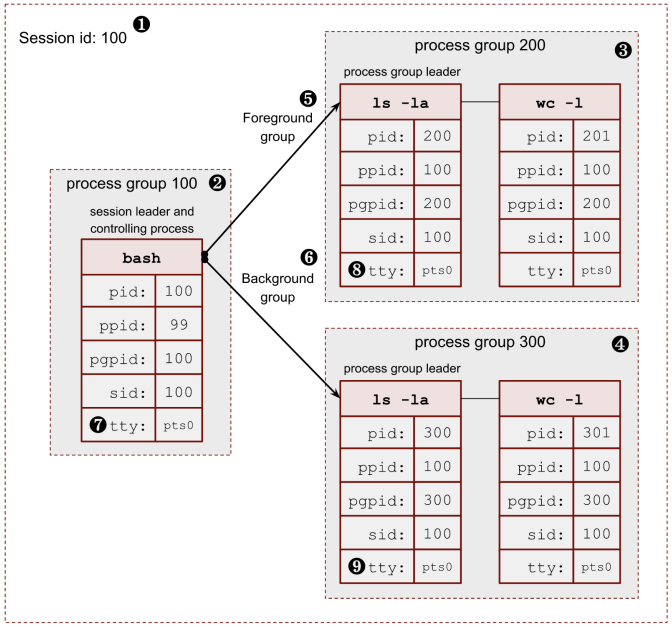
[232] https://medium.com/@boutnaru/linux-memory-management-part-2-demand-paging-38afdd85f447
[233] https://elixir.bootlin.com/linux/v6.0.1/source/include/linux/sched.h#L1030
[234] https://elixir.bootlin.com/linux/v6.0.1/source/mm/memory.c#L5077

Session

In general, a “Session” is a collection of “Process Groups”[235], which is identified by a session ID (SID). This value is inherited from the session leader that has created the session[236]. It is common for services/daemons to be encapsulated in a session, which simplifies stopping/starting them by grouping all of their processes[237].

Moreover, a new session is created in two major cases. First, when logging with a user interactively the shell processes becomes a session leader. Second, a daemon starts and wants to create its own session. A diagram is included below which demonstrates the relationship between the sessions and process groups [238].

Lastly, in order to create a new session we can use the “setsid” system call[239]. Also, we can read the session ID of a process from “/proc/[PID]/status”, it is saved in the “NSsid” field[240]. Overall, a session can have a controlling tty and at most one process group in the foreground[241].



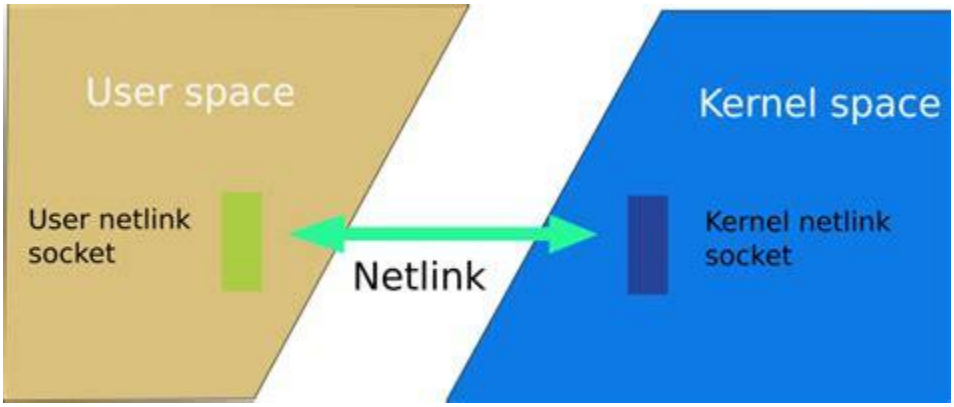
[235] https://medium.com/@boutnaru/the-linux-concept-journey-process-group-b39e733087e9
[236] https://biriukov.dev/docs/fd-pipe-session-terminal/3-process-groups-jobs-and-sessions/
[237] https://www.elastic.co/blog/linux-process-and-session-model-as-part-of-security-alerting-and-monitoring
[238] https://biriukov.dev/docs/fd-pipe-session-terminal/3-process-groups-jobs-and-sessions/
[239] https://man7.org/linux/man-pages/man2/setsid.2.html
[240] https://man7.org/linux/man-pages/man5/proc_pid_status.5.html
[241] https://www.win.tue.nl/~aeb/linux/lk/lk-10.html

IPC Methods Between Kernel and User Space

Due to the fact that kernel-mode code and user-mode code don't share the same memory address space (because of security reasons) in case we want them to communicate we need some kind of an "IPC" mechanism. Examples of such methods for passing data between user-mode and kernel mode are: system calls (like ioctl), the proc filesystem and netlink sockets[242].

Overall, system calls are a fundamental interface between user-mode code and the Linux kernel[243]. Among the different syscalls we have "ioctl" (I/O Control), which can be used for manipulating the underlying device of special file[244].

Lastly, we use the proc filesystem for reading/writing (like by using the read/write syscalls) information to the kernel[245]. By creating a proc kernel module we can expand the IPC flow of information between user mode and kernel mode[246]. Also, we can use the socket API (socket related syscalls) for IPC between user-mode and kernel mode by leveraging netlink sockets[247] - as shown in the diagram below[248].



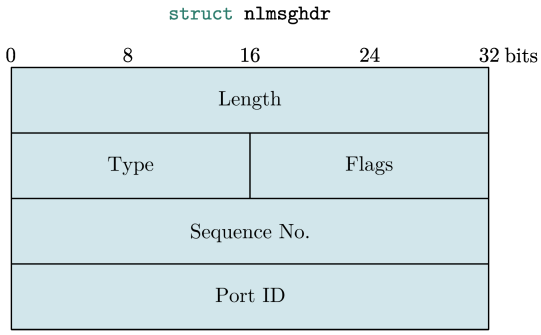
[242] https://github.com/mwarning/netlink-examples/blob/master/articles/Why_and_How_to_Use_Netlink_Socket.md
[243] https://medium.com/@boutnaru/the-linux-concept-journey-syscalls-system-calls-efcd7703e072
[244] https://man7.org/linux/man-pages/man2/ioctl.2.html
[245] https://man7.org/linux/man-pages/man5/procfs.5.html
[246] https://www.cs.fsu.edu/~cop4610t/lectures/project2/procfs_module/proc_module.pdf
[247] https://man7.org/linux/man-pages/man7/netlink.7.html
[248] https://dev.to/jemaloqiu/netlink-communication-between-kernel-and-user-space-2mg1

Netlink

Netlink is a socket family (datagram based) that we can use for IPC[249] between user-space and kernel space components. Those sockets can be used for local communication only by leveraging the socket API (rec/recvmsg/etc). Netlink is often used as a replacement for ioctl[250]. While unix domain sockets leverage the file-system namespace, Netlink sockets are usually addressed using process identifiers[251].

Overall, in order to use netlink we need to pass "AF_NETLINK" as the "communication domain" as the first argument of the "socket" syscall[252]. The message header of netlink is defined using "struct nlmsghdr" as part of "netlink.h" in the Linux kernel source code[253] - as shown in the diagram below. This is followed by a protocol header, like "Generic Netlink" header (struct genlmsghdr)[254].

Lastly, the third argument (netlink_family) for the socket syscall when using Netlink is used for selecting the relevant kernel module\netlink group we want to communicate with. Examples of such values are: "NETLINK_ROUTE" (receives routing and link updates), "NETLINK_XFRM" (IPSec), "NETLINK_AUDIT" (auditing), "NETLINK_RDMA" (Infiniband RDMA) and more[255]. By the way, we can add our own Netlink handler and implement additional Netlink protocols[256].



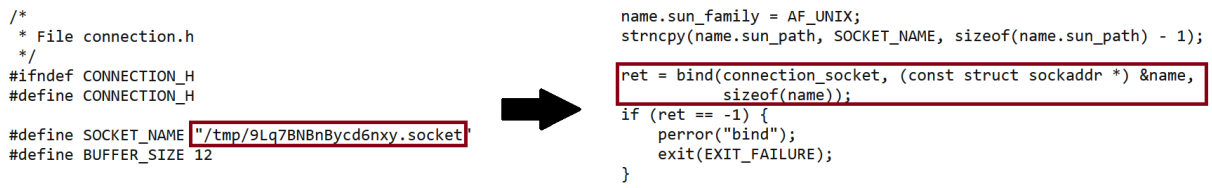
[249] https://medium.com/@boutnaru/the-linux-concept-journey-ipc-methods-between-kernel-and-user-space-3cde144341e9
[250] https://docs.kernel.org/userspace-api/netlink/intro.html
[251] https://en.wikipedia.org/wiki/Netlink
[252] https://man7.org/linux/man-pages/man2/socket.2.html
[253] https://elixir.bootlin.com/linux/v6.13.5/source/include/uapi/linux/netlink.h#L52
[254] https://www.yaroslavps.com/weblog/genl-intro/
[255] https://man7.org/linux/man-pages/man7/netlink.7.html
[256] https://www.linuxjournal.com/article/7356

Unix Domain Sockets

“Unix Domain Sockets” are a method for local IPC (Inter Process Communication). They are based on the socket family “AF_UNIX” (aka “AF_LOCAL”). Also, unix domain sockets (UDS) can be unnamed\bound to a filesystem location\path (based on the information passed to the “bind” syscall or when using the "socketpair" syscall) - as shown in the screenshot below. Unix domain sockets allows us to pass FDs (file descriptors) or credentials using ancillary data[257] by leveraging the “sendmsg” and “recvmsg” syscalls[258].

Overall, unix domain sockets don’t reorder datagrams and from kernel version 2.6.4 we also have “SOCK_SEQPACKET”[259] for sequential packet socket. One of the benefits of using unix domain sockets is we can leverage the socket API (send/recv/connect/bind/listen/accept/etc) and its error handling[260] as opposed of using a pipe/named pipe for IPC[261].

Lastly, UDS provides different advantages such as: bypassing the networking stack which improves performance detecting error as compared to UDP\pipes[262]. Although it is not mandatory, socket files usually have the “.sock”\”.socket” extension like “docker.sock”[263].



```
/*
 * File connection.h
 */
#ifndef CONNECTION_H
#define CONNECTION_H

#define SOCKET_NAME "/tmp/9Lq7BNBnBycd6nxy.socket"
#define BUFFER_SIZE 12
```

```
name.sun_family = AF_UNIX;
strncpy(name.sun_path, SOCKET_NAME, sizeof(name.sun_path) - 1);

ret = bind(connection_socket, (const struct sockaddr *) &name,
           sizeof(name));
if (ret == -1) {
    perror("bind");
    exit(EXIT_FAILURE);
}
```

[257] https://man7.org/linux/man-pages/man7/unix.7.html
[258] https://en.wikipedia.org/wiki/Unix_domain_socket
[259] https://elixir.bootlin.com/linux/v6.13.7/source/include/linux/net.h#L55
[260] https://dev.to/____marcell/unix-domain-socketipc-how-to-make-two-programs-communicate-with-each-other-8l6
[261] https://man7.org/linux/man-pages/man3/mkfifo.3.html
[262] https://docs.datadoghq.com/developers/dogstatsd/unix_socket/
[263] https://www.bordergate.co.uk/docker-penetration-testing/

IOCTL (Input/Output Control)

IOCTL (Input/Output Control) is a syscall as part of the Linux operating system[264] that is used for manipulating parameters of devices exposed by special files. For example terminals, which are character devices[265] can be controlled using "ioctl"[266].

Overall, like with other syscalls we can user "ioctl" as an IPC mechanisms for passing information between user-mode and kernel mode[267]. The first argument of the syscall should be an open fd (file descriptor) ,the second one is a device-dependent request code and the third untyped pointer to memory[268]. An example of request code (like SIOCDELRT) used for IP routing control is shown in the table below[269]. We can check the Linux source code for such implementations[270].

Lastly, "ioctl" is used for input/output operations in which we can leverage other syscalls (like read/write/etc). It is a concept which is relevant also for other operating systems (not only Linux)[271]. For example, in Windows we have "DeviceIoControl"[272].

ioctl Function (Cont.)

Category	request	Description
interface	SIOCGIFCONF	get list of all interfaces
	SIOCSIFADDR	set interface address
	SIOCGIFADDR	get interface address
	SIOCSIFFLAGS	set interface flags
	SIOCGIFFLAGS	get interface flags
	SIOCSIFDSTADDR	set point-to-point address
	SIOCGIFDSTADDR	get point-to-point address
	SIOCGIFBRDADDR	get broadcast address
	SIOCSIFBRDADDR	get broadcast address
	SIOCGIFNETMASK	get subnet mask
	SIOCSIFNETMASK	set subnet mask
	SIOCGIFMETRIC	get interface metric
	SIOCSIFMETRIC	set interface metric
	SIOC…	(many more; implementation dependent)
routing	SIOCADDRT	add route
	SIOCDELRT	delete router

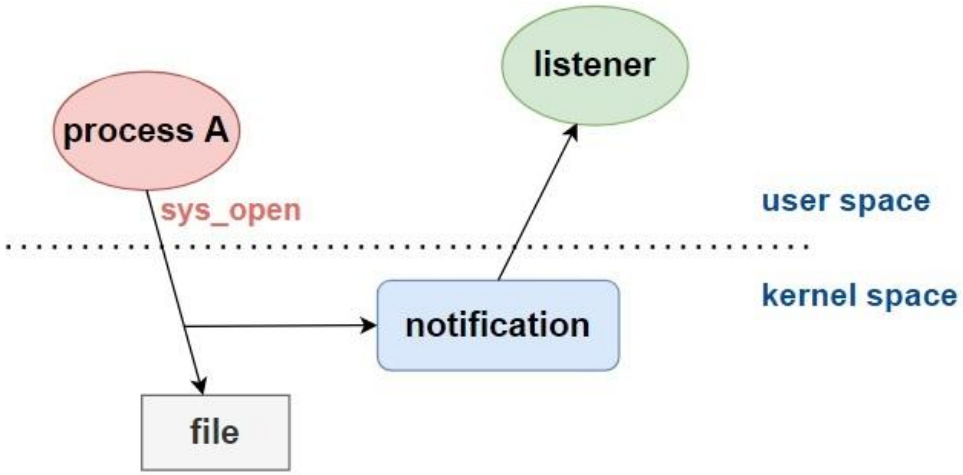
[264] https://medium.com/@boutnaru/the-linux-concept-journey-syscalls-system-calls-efcd7703e072
[265] https://medium.com/@boutnaru/the-linux-concept-journey-character-devices-0c75aa70ceb2
[266] https://man7.org/linux/man-pages/man2/ioctl.2.html
[267] https://medium.com/@boutnaru/the-linux-concept-journey-ipc-methods-between-kernel-and-user-space-3cde144341e9
[268] https://linux.die.net/man/2/ioctl
[269] https://www.slideserve.com/penda/ioctl-function
[270] https://elixir.bootlin.com/linux/v6.13.7/source/net/ipv4/fib_frontend.c#L634
[271] https://en.wikipedia.org/wiki/Ioctl
[272] https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol

dnotify (Directory Notification)

dnotify (Directory Notification) is a file system event monitoring capability as part of the Linux kernel. It has been introduced as part of kernel 2.4. It has several limitations. First, dnotify can only watch directories. Second, it requires maintaining an open fd (file descriptor) to the directory that the user wants to watch. Thus, we can't unmount a device which has a monitored directory and our watch list is bound by the open file limit of the specific process performing the monitoring[273].

Overall, we can use dnotify for being notified when a directory\files in it are changed. The notifications are registered using the "fcntl" syscall[274] while the events are being delivered using signals. The relevant events are: a file in the directory was accessed (read), a file in the directory was modified, a file in the directory was unliked, a file in the directory was renamed, a file was created in the directory and a file in the directory had its attributes changed[275].

Lastly, dnotify had been replaced by inotify[276]. We can also checkout the implementation of dnotify as part of the Linux kernel[277]. When a specific syscall is triggered (which is relevant for an event we want to monitor) the kernel can notify (using signals) about the operation - as shown in the diagram below[278].



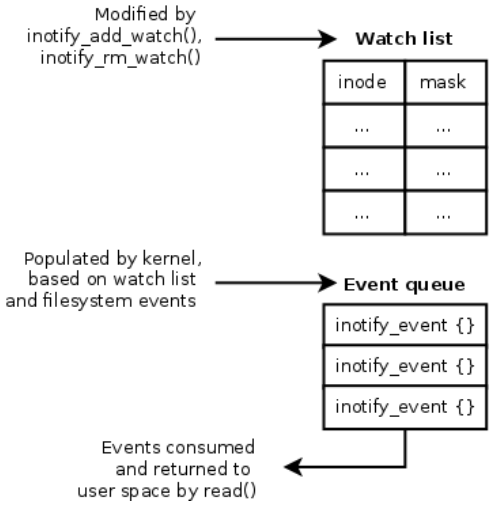
[273] https://en.wikipedia.org/wiki/Dnotify
[274] https://man7.org/linux/man-pages/man2/fcntl.2.html
[275] https://www.kernel.org/doc/Documentation/filesystems/dnotify.txt
[276] https://man7.org/tlpi/code/online/dist/inotify/dnotify.c.html
[277] https://elixir.bootlin.com/linux/v6.13.7/source/fs/notify/dnotify/dnotify.c
[278] https://jailbreakfox.github.io/2022/04/22/Linux%E6%96%87%E4%BB%B6%E4%BA%8B%E4%BB%B6%E7%9B%91%E6%8E%A7/

inotify (Inode Notification)

inotify (Inode Notification) is a simple change notification system[279]. The Linux kernel supports inotify since kernel 2.6.13. Using the inotify API we can monitor individual files/folders. inotify works based on inodes[280]. Thus if we monitor a file an event can be triggered for an activity on a link also[281]. There are several configuration limits for inotify: "max_user_instances" (how many applications can watch files, per user), "max_user_watches" (how many filesystem items can be watched, across all applications, per user) and "max_queued_events" (how many filesystem events will be held in the kernel queue if the application does not read them). Those are in parallel to physical limits[282].

Overall, the above configuration is read\altered by using procfs (/proc/sys/fs/inotify/) and/or sysctl[283]. Also, inotify had replaced dnotify[284] due to reasons such as: inotify supports monitoring individual files and entire directories, inotify uses fewer resources, inotify is more widely supported and provides more detailed event notifications[285].

Lastly, inotify uses dedicated syscalls: "inotify_init"\"inotify_init1" for initializing an inotify instance[286], "inotify_add_watch" used for adding a watch to an initialized inotify instance[287] and "inotify_rm_watch" that removes an existing watch[288]. The events are being consumed by using the "read" syscall - as shown in the diagram below[289].



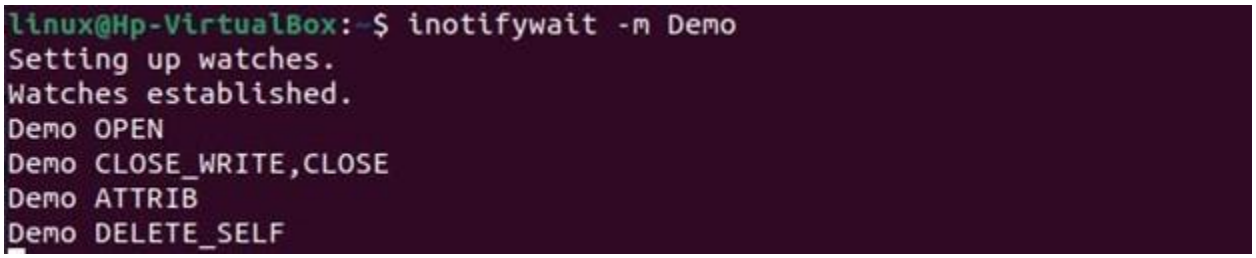
[279] https://www.kernel.org/doc/html/v6.4/filesystems/inotify.html
[280] https://medium.com/@boutnaru/linux-what-is-an-inode-7ba47a519940
[281] https://man7.org/linux/man-pages/man7/inotify.7.html
[282] https://watchexec.github.io/docs/inotify-limits.html
[283] https://www.suse.com/support/kb/doc/?id=000020048
[284] https://medium.com/@boutnaru/the-linux-concept-journey-dnotify-directory-notification-6204db62ea45
[285] https://www.baeldung.com/linux/dnotify-inotify-monitor-directory
[286] https://man7.org/linux/man-pages/man2/inotify_init.2.html
[287] https://man7.org/linux/man-pages/man2/inotify_add_watch.2.html
[288] https://man7.org/linux/man-pages/man2/inotify_rm_watch.2.html
[289] https://lwn.net/Articles/604686/

Limitations When Using inofity (Inode Notification)

As with every technology also “inotify” has its own limitations. Among those limitations we find the following. Inability to monitor recursive directories. Thus, we need to have a separate “inotify watch”[290] for every subdirectory we want to monitor. Also, in case a file is renamed we get two discrete events that we have to correlate (there is not a specific rename event), which can have a potential race condition[291].

Moreover, the “inotify” API does not provide any information about the process/user which caused the creation of an inotify event. Hence, in case we use the “inotifywatch” utility only the operation is given as extra metadata[292] - as shown in the screenshot below[293]. By the way, inotify events are based on filenames, due to that we can get an event which is processed after the filename has been changed[294].

Lastly, because inotify is based on an event queue in case of an overflow, events can be lost[295]. It is important to know that there are some filesystems which don’t fully support inotify such like a couple of network filesystems and procfs[296].



[290] https://medium.com/@boutnaru/the-linux-concept-journey-inotify-inode-notification-a514cccee704
[291] https://www.clariontech.com/blog/all-you-need-to-know-about-inotify
[292] https://linux.die.net/man/1/inotifywait
[293] https://linuxhint.com/linux-inotify-command/
[294] https://linux.die.net/man/7/inotify
[295] https://groups.google.com/g/lsyncd/c/2dB_z8K5OSs
[296] https://inotify.aiken.cz/?section=inotify&page=faq&lang=en

fanotify (Inode Notification)

“fanotifiy” is a Linux API that can be used by user space applications in order to receive file events in the system in real-time[297]. Examples of relevant use-cases are: virus scanning and\or hierarchical storage management. Until kernel 5.1 there was no support for create\delete\move events as part of “fanotify”[298]. Thus, before kernel 5.1 we would need to use “inotify”[299] or the older “dnotify”[300].

Overall, as compared to “inotify” we can use “fanotify” to monitor all of the objects in a mounted filesystem. Also, it has the ability to make access permission decisions and to read\modify files before access by other applications. “fanotify” is based on dedicated syscalls (like “fanotify_init” and “fanotify_mark”), general syscalls (like “read” and “write”) and “notifications groups”[301] - more on those and others in future writeups.

Lastly, we can check out the “fanotify” implementation as part of the Linux kernel source code[302]. While “inotify” does not contain any information about the process which triggered an event, “fanotify” has the PID of that process[303]. For enabling (fanotify) file-system wide access notification we should set “CONFIG_FANOTIFY” as part of kernel build configuration[304]. For fanotify permissions checking we should enable “CONFIG_FANOTIFY_ACCESS_PERMISSIONS”[305] - as shown in the screenshot below.

```
# SPDX-License-Identifier: GPL-2.0-only
config FANOTIFY
        bool "Filesystem wide access notification"
        select FSNOTIFY
        select EXPORTFS
        default n
        help
           Say Y here to enable fanotify support.  fanotify is a file access
           notification system which differs from inotify in that it sends
           an open file descriptor to the userspace listener along with
           the event.

           If unsure, say Y.

config FANOTIFY_ACCESS_PERMISSIONS
        bool "fanotify permissions checking"
        depends on FANOTIFY
        default n
        help
           Say Y here is you want fanotify listeners to be able to make permissions
           decisions concerning filesystem events.  This is used by some fanotify
           listeners which need to scan files before allowing the system access to
           use those files.  This is used by some anti-malware vendors and by some
           hierarchical storage management systems.

           If unsure, say N.
```

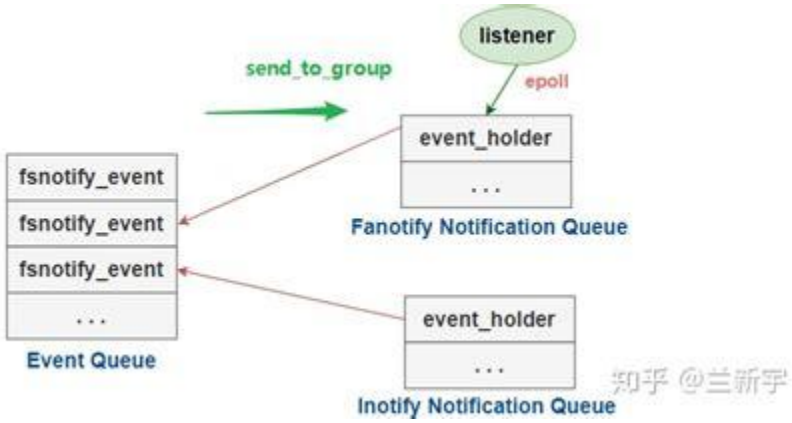
[297] https://success.trendmicro.com/en-US/solution/KA-0014210
[298] https://manpages.ubuntu.com/manpages/focal/en/man7/fanotify.7.html
[299] https://medium.com/@boutnaru/the-linux-concept-journey-inotify-inode-notification-a514cccee704
[300] https://medium.com/@boutnaru/the-linux-concept-journey-dnotify-directory-notification-6204db62ea45
[301] https://man7.org/linux/man-pages/man7/fanotify.7.html
[302] https://elixir.bootlin.com/linux/v6.13.7/source/fs/notify/fanotify
[303] https://habr.com/en/companies/pt/articles/783038/
[304] https://cateee.net/lkddb/web-lkddb/FANOTIFY.html
[305] https://elixir.bootlin.com/linux/v6.13.7/source/fs/notify/fanotify/Kconfig

fsnotify (File System Notification)

fsnotify (File System Notification) is a generic framework used to hook in order to provide filesystem notification[306]. It had been created to reduce in-source duplication from both "dnotify"[307], "inotify"[308] and "fanotify"[309].

Overall, inotify was largely rewritten in 2009 to make use of the fsnotify infrastructure[310]. The same was done also for "dnotify"[311]. Based on the kernel source code it seems that "fanotify" is based on "fsnotify" since it was merged to the main kernel as part of version 2.6.36[312] - as shown in the screenshot below[313].

Lastly, the relevant kernel configuration for "fsnotify" is "CONFIG_FSNOTIFY"[314]. Also, the VFS (Virtual File System) layer calls hook specific functions (linux/fsnotify.h) which in turn call the "fsnotify" function. It calls out to all of the registered "fsnotify_group", those groups can then use the notification event[315].



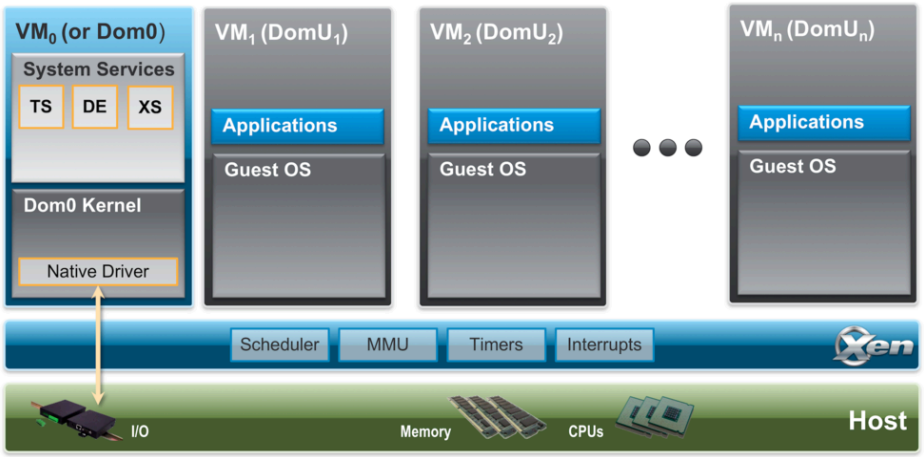
[306] https://elixir.bootlin.com/linux/v6.14.1/source/include/linux/fsnotify.h#L6
[307] https://medium.com/@boutnaru/the-linux-concept-journey-dnotify-directory-notification-6204db62ea45
[308] https://medium.com/@boutnaru/the-linux-concept-journey-inotify-inode-notification-a514cccee704
[309] https://medium.com/@boutnaru/the-linux-concept-journey-fanotify-inode-notification-6be8775cff9a
[310] https://elixir.bootlin.com/linux/v6.14.1/source/fs/notify/inotify/inotify_fsnotify.c#L13
[311] https://elixir.bootlin.com/linux/v6.14.1/source/fs/notify/dnotify/dnotify.c#L8
[312] https://elixir.bootlin.com/linux/v2.6.36/source/fs/notify/fanotify/fanotify.c#L3
[313] https://blog.csdn.net/weixin_32842805/article/details/112274768
[314] https://www.kernelconfig.io/config_fsnotify
[315] https://elixir.bootlin.com/linux/v6.14.3/source/fs/notify/fsnotify.c#L517

Xen Hypervisor

Xen is a type-1 hypervisor which is open source. Among its key features are: small footprint, driver isolation (while the main device driver for a system to run inside of a virtual machine and in case of a crush/compromise it can be restarted without affecting the rest of the system) and paravirtualization support (Xen can run on hardware that does not have virtualization extensions). It was originally developed by the University of Cambridge Computer Laboratory. Today it is being developed by the Linux Foundation[316].

Overall, the Xen hypervisor is executed just after the bootloader and on top of it different virtual machines (VMs) can run. A running VM is called domain/guest - as shown in the diagram below. There is a special VM called "Domain 0" aka the "Control Domain" which has special privileges (access the hardware directly, handles all access to the system's I/O and interacts with the VMs)[317].

Lastly, Xen supports ARM\x86-64\IA-32 processors. We can download Xen directly from the XenProject website[318]. I suggest also going over Xen's source code for more information[319].



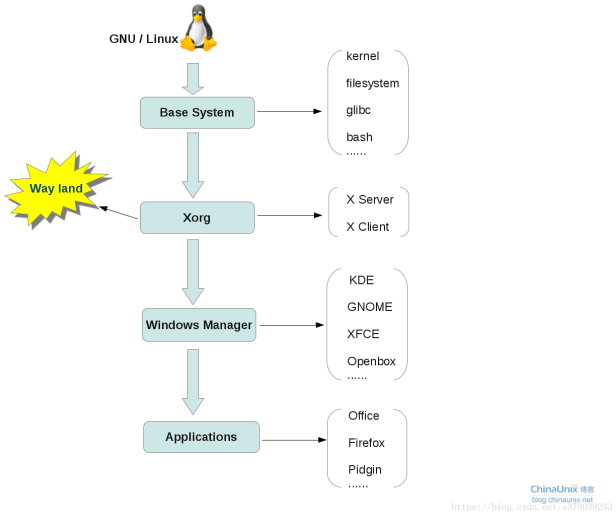
[316] https://en.wikipedia.org/wiki/Xen
[317] https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview#What_is_the_Xen_Project_Hypervisor?
[318] https://downloads.xenproject.org/release/xen/
[319] https://github.com/xen-project/xen

Xorg (X Windows System)

Xorg (x.org aka X) is an open source project which provides an implementation of the "X Window System" (X11). The development work is done together with the "freedesktop.org"[320] community. It is the most popular display server among Linux users. By the way, "Wayland" is a replacement for the X11 windows system protocol[321] - as shown in the diagram below[322].

Overall, Xorg is a full featured X server that was originally designed for UNIX and UNIX-like operating systems (running on Intel x86 hardware). Today it runs on a wider range of hardware (such as Compaq Alpha, Intel IA64, AMD64, SPARC and PowerPC) and OS platforms (like FreeBSD, NetBSD, OpenBSD, and Solaris). Xorg supports two major connection types: "Local" (for example by leveraging Unix-domain socket) and ""TCP/IP" in which listens on port "6000+n" where "n" is the display number[323].

Lastly, Xorg is an open-source application that interacts with client applications through the X11 protocol. The X system (debuted in 1984) was designed to render graphics over a network. Also, it is based on the client/server model. The benefit of the client/server model is that it allows clients to run either locally\remotely. When a client connects and sends inputs like mouse movements or keystrokes, it draws on the window display. Due to that client/server model, applications need to communicate with Xorg before the compositor can create the window[324].



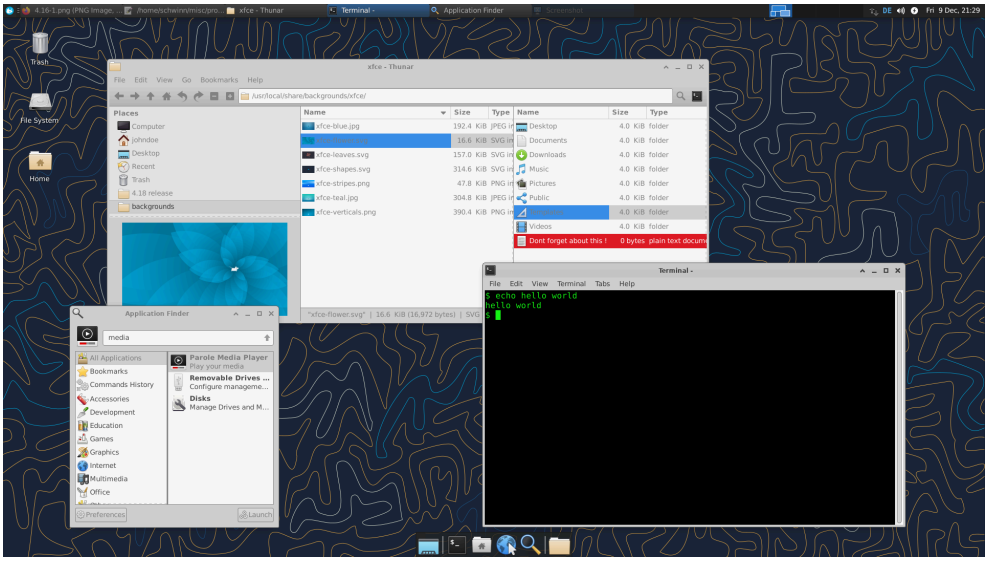
[320] https://www.freedesktop.org/wiki/
[321] https://wiki.archlinux.org/title/Xorg
[322](https://blog.csdn.net/a379039233/article/details/80782351
[323] https://man.archlinux.org/man/Xorg.1
[324] https://www.cbtnuggets.com/blog/technology/devops/wayland-vs-xorg-wayland-replace-xorg

Xfce (Desktop Environment)

Xfce is a lightweight desktop environment for UNIX-like operating systems. It is very low on system resources, without giving up on a visually appealing and user friendly[325] - as shown in the screenshot below[326]. The Xfce desktop environment was initially released in 1996[327].

Overall, as with GNOME it is also based on the GTK toolkit (but we don't need to think about Xfce as a fork of GNOME)[328]. Xfce is the default desktop environment in various Linux distributions. Examples are: "Xubuntu"[329], "Linux Lite"[330], "QubesOS"[331], "Kali Linux"[332] and "Whonix"[333].

Lastly, Xfc comes built in with different components such as: "Catfish" (desktop search), "Clipman" (clipboard manager), "Mousepad" (text editor), "Thunar" (file manager), xfce4-Panel (task manager), "Xfwm" (window manager), Xfburn (burning CD/DVD/BRD) and more[334]. We can check out Xfce core components and applications source code for more information/details[335].



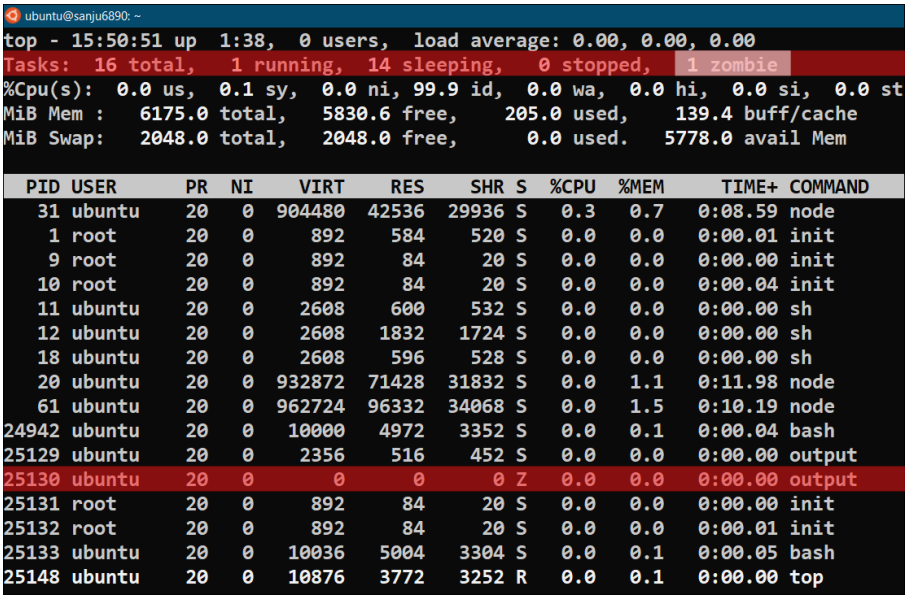
[325] https://www.xfce.org/
[326] https://www.xfce.org/about/screenshots
[327] https://betawiki.net/wiki/Xfce3
[328] https://en.wikipedia.org/wiki/Xfce
[329] https://xubuntu.org/
[330] https://www.linuxliteos.com/
[331] https://www.qubes-os.org/
[332] https://www.kali.org/
[333] https://www.whonix.org
[334] https://wiki.debian.org/Xfce
[335] https://gitlab.xfce.org/xfce

Zombie Processes

In general, zombie processes (aka defunct processes) are processes which have finished their execution flow but still have an entry in the overall processes table. This phenomena happens due to the fact that the parent process has not read the child's exit status. Zombie processes don't require additional resources, however the existence of them can clutter the process table[336].

Moreover, in case we have zombie processes we can't send them a "SIGKILL", thus we need to signal the parent process to read the child process exit status. If we use "ps" zombie processes are marked as "<defunct>" and in the state column ("STAT or "S" header) the are denoted using "Z"[337]. Also, in the top command there is a specific counter for zombie processes[338] - as shown in the screen below[339].

Lastly, the zombie state is saved as part of the the "exit_state" field[340] of the process' "struct task_struct"[341] structure. It is done using the "EXIT_ZOMBIE" define value[342].



[336] https://ioflood.com/blog/ps-linux-command/
[337] https://man7.org/linux/man-pages/man1/ps.1.html
[338] https://man7.org/linux/man-pages/man1/top.1.html
[339] https://www.linuxfordevices.com/tutorials/linux/defunct-zombie-process
[340] https://elixir.bootlin.com/linux/v6.6.8/source/include/linux/sched.h#L879
[341] https://medium.com/@boutnaru/linux-kernel-task-struct-829f51d97275
[342] https://elixir.bootlin.com/linux/v6.6.8/source/include/linux/sched.h#L93

Uninterruptible Process

In the Linux realm we have two types of waiting processes: “interruptible processes” and “uninterruptible processes”. In general this type of processes are waiting for an event of a resource[343]. An “Uninterruptible” process (called task in the source code of the Linux kernel, thus a “Uninterruptible Task”) which is executing a system call that cannot be interrupted by a signal[344].

Moreover, “uninterruptible process” is flagged as “D” state in output of the “ps” utility[345], described as “uninterruptible sleep (usually I/O)” - as shown in the screenshot below[346]. Also, this type of processes are defined in the Linux kernel source code as “TASK_UNINTERRUPTIBLE”[347]. As of kernel version 6.10 “TASK_UNINTERRUPTIBLE” is referenced in 248 files across the Linux source code[348].

Lastly, “uninterruptible process” is not expecting to be woken up by anything other than whatever it is waiting for, either because it cannot easily be restarted, or because programs are expecting the system call to be atomic[349]. Also, we can’t kill such processes, even if we use SIGKILL[350].

```
root@localhost:~# ps aux | grep /nfs/a
root     28794  0.0  0.0   7308   692 pts/4    D+   11:22   0:00 cat /mnt/nfs/a
root     29286  0.0  0.0  14224   972 pts/3    S+   11:22   0:00 grep --color=auto /nfs/a
root@localhost:~#
```

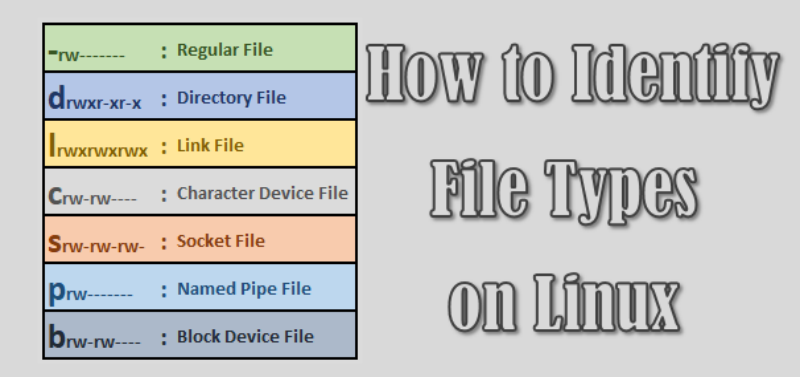
[343] https://www.science.unitn.it/~fiorella/guidelinux/tlk/node45.html
[344] https://medium.com/@boutnaru/the-linux-concept-journey-signals-d1f37a9d2854
[345] https://man7.org/linux/man-pages/man1/ps.1.html
[346] https://juicefs.com/zh-cn/blog/engineering/howto-solve-d-status-process/
[347] https://elixir.bootlin.com/linux/v6.10/source/include/linux/sched.h#L97
[348] https://elixir.bootlin.com/linux/v6.10/C/ident/TASK_UNINTERRUPTIBLE
[349] https://stackoverflow.com/questions/223644/what-is-an-uninterruptible-process
[350] https://www.suse.com/support/kb/doc/?id=000016919

Linux File Types

As we know the philosophy of Linux is that “Everything is a file”. However, not all files are created equally. We have 7 different file types: directory, regular file, named pipe, socket, symbolic link, block device file and character device file[351] - more information about each type in future writeups.

Overall, we can identify the type of a file using the “ls” utility using the “-l” argument[352]. The first character in each line identifies the type of the file - as described in the table below[353].

Lastly, we can display only specific file types by filtering the output of “ls” using “grep”[354]. For example if we want to see all regular files in the current directory we can use the following one-liner: “ls -la | grep ^-”[355].



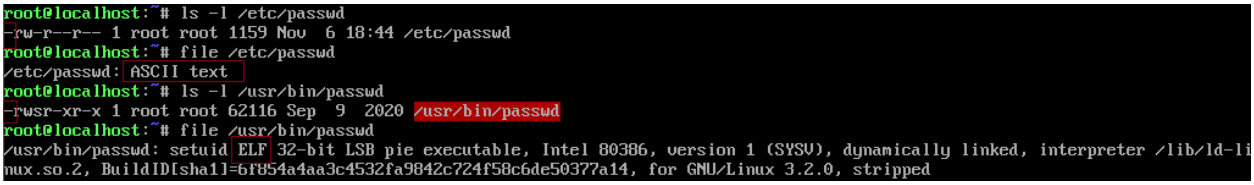
[351] https://linuxconfig.org/identifying-file-types-in-linux
[352] https://man7.org/linux/man-pages/man1/ls.1.html
[353] https://www.2daygeek.com/wp-content/uploads/2019/01/find-identify-file-types-in-linux-4.png
[354] https://man7.org/linux/man-pages/man1/grep.1.html
[355] https://www.2daygeek.com/find-identify-file-types-in-linux/

Regular File

As we know the philosophy of Linux is that “Everything is a file”. However, not all files are created equally. As you know there are seven different file types used in Linux[356]. The following writeup is going to focus on regular files, those that when using “ls -l”[357] are marked with “-” - as shown in the screenshot below (taken using https://copy.sh/v86/?profile=archlinux).

Overall, it is important to understand that a type of a “regular file” is from the perspective of the operating system and it does not describe the format of the file itself. We can check the format of a file using the “file” command[358]. For example a “Regular File” can be an image file (such as PNG, gif, JPEG or BMP), a text file, an executable (like ELF or PE), archive files (like RAR and ZIP) and more - as shown in the screenshot below.

Lastly, probably the easiest way to create a regular file is by using the “touch” command[359]. By the way, we can create a regular file in any directory[360]. For that to happen we just need to have write permissions to the directory itself. Also, the mounted filesystem should not be read-only.



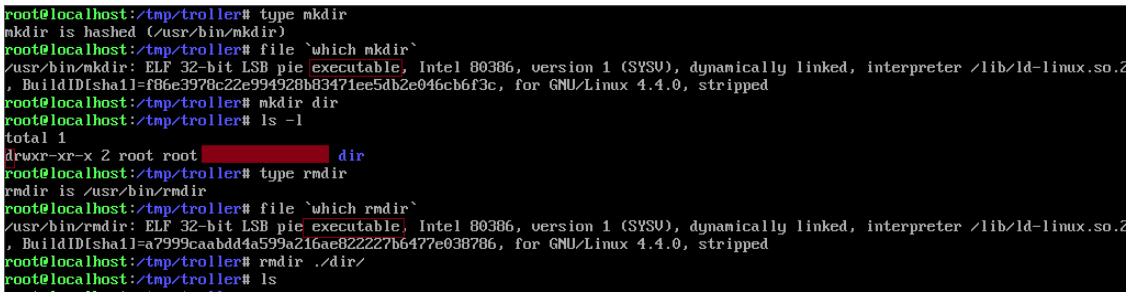
[356] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[357] https://man7.org/linux/man-pages/man1/ls.1.html
[358] https://man7.org/linux/man-pages/man1/file.1.html
[359] https://man7.org/linux/man-pages/man1/touch.1.html
[360] https://www.geeksforgeeks.org/how-to-find-out-file-types-in-linux/

Directory File

As you know there are seven different file types used in Linux[361]. Among them we have a "directory" file type, we can think about it as a "file" that holds in its content file names and there representing inode numbers[362].

Thus, using the "rm" utility[363] basically removes the file from the directory and does not delete it (until the reference count equals "0"). Also, because of that removing a file does not require any permissions on the file itself, it requires having "write permissions" to the directory containing the file.

Lastly, in order to create a new directory we can use the "mkdir" utility[364] in order to create a new directory or the "rmdir" utility[365] in order to remove a directory - as shown in the screenshot below (taken using https://copy.sh/v86/?profile=archlinux). A directory is marked with a "d" as the first character in the output of the "ls -l" command[366] - also shown in the screenshot below.



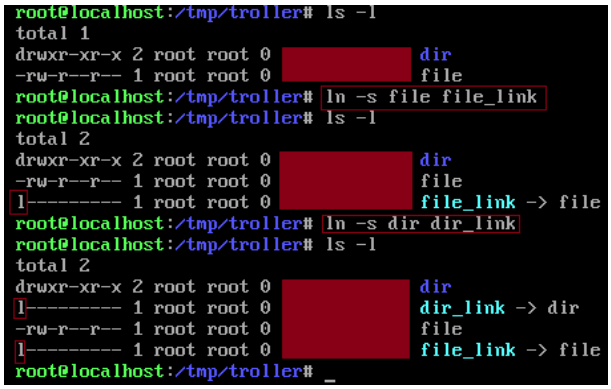
[361] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[362] https://medium.com/@boutnaru/linux-what-is-an-inode-7ba47a519940
[363] https://linux.die.net/man/1/rm
[364] https://man7.org/linux/man-pages/man1/mkdir.1.html
[365] https://man7.org/linux/man-pages/man1/rmdir.1.html
[366] https://man7.org/linux/man-pages/man1/ls.1.html

Link File (aka Symbolic Link)

As you know there are seven different file types used in Linux[367]. Among them we have a “link” file type (aka “symbolic link’), which is used for pointing to other files[368]. When using “ls -l”[369] link files are marked with “l” in the output - as shown in the screenshot below.

Overall, we can use the “ln” command[370] to make links between files. It supports both creating “hard links”[371] and “symbolic links”. In order to create a “link file” we need to use the “-s” switch - as shown in the screenshot below.

Lastly, as opposed to “hard links” that can’t be created to a directory a “symbolic link” can point to a directory - as shown in the screenshot below. By the way, “symbolic links” are also called “soft links”[372].



[367] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[368] https://www.freecodecamp.org/news/linux-ln-how-to-create-a-symbolic-link-in-linux-example-bash-command/
[369] https://man7.org/linux/man-pages/man1/ls.1.html
[370] https://www.man7.org/linux/man-pages/man1/ln.1.html
[371] https://medium.com/@boutnaru/the-linux-concept-journey-hard-link-f3e9b3d6b8c4
[372] https://www.geeksforgeeks.org/soft-hard-links-unixlinux/

Socket File

A socket file is one of the file types supported under Linux[373]. The main goal of a socket file is to pass information between applications. For convenient applications which create socket files use the ".socket"/"*.sock" suffix for the file name. Examples for such file names are: acpid.socket and docker.sock[374].

Overall, we can use the "AF_UNIX"/"AF_LOCAL" socket family for communicating between processes on the same machine. This type of socket is known as "Unix Domain Sockets". This type of sockets can be unnamed or to be bound to a file-system path. We can use both "SOCK_STREAM" (for stream oriented socket) or "SOCK_DGRAM" (for data-gram oriented). Since Linux 2.6.4 we can use "SOCK_SEQPACKET" for a sequenced-packet socket, that delivers messages in the order that they were sent[375].

Lastly, when we create such a socket a "socket file" is created - as shown in the screenshot below. Due to the fact it is a socket we can use any of the socket API functions on[376] - it is a big advantage as opposed to using regular files as an IPC. Also, it is important to understand that even if "Unix Domain Sockets" are designed for local communication only, if we manage to create our socket file on a remote device (like by using NFS/SMB) we can pass information between remote processes - it can also be done using regular files however it is not the preferred method for that.

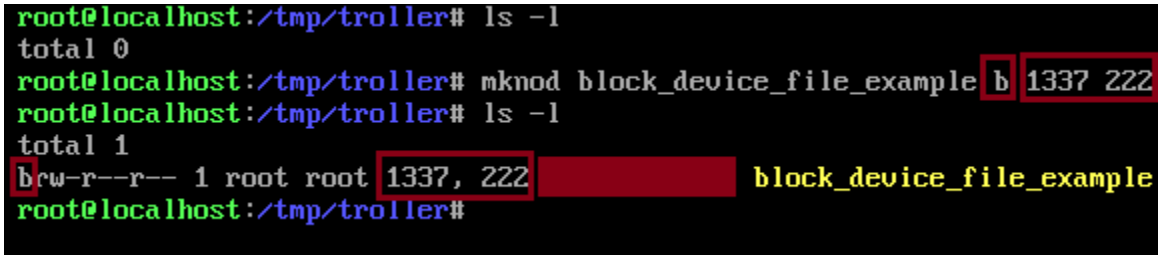
[373] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[374] https://www.bogotobogo.com/Linux/linux_File_Types.php
[375] https://man7.org/linux/man-pages/man7/unix.7.html
[376] https://www.baeldung.com/linux/python-unix-sockets

Block Device File

As we know the philosophy of Linux is that “Everything is a file”. However, not all files are created equally. A block device file is one of the supported file types by Linux[377]. The goal of a “Block Device File” is to provide buffered access to a hardware device[378].

Overall, block devices provide the ability to randomly access data which is organized in fixed-size blocks[379]. Also, block device files are flagged with “b” in the output of “ls -l”[380]. By default we can find them in the “/dev” directory - as shown in the screenshot below. Examples of such block device files are: “/dev/sda” (first identified SCSI storage device), “/dev/mmcblk2” (the third identified SD/MMC/eMMC storage device) and “/dev/nvme5n1” (the first identified device on the sixth controller).

Lastly, we can create a block device file using the “mknod” command[381], when using the command we need to provide the major and minor numbers of the device[382] - as shown in the screenshot below. By the way, we can call the “mknod”/”mknodat” syscall[383] in order to create a block device file. Another useful command is “lsblk”[384] which can list block devices on a specific system.



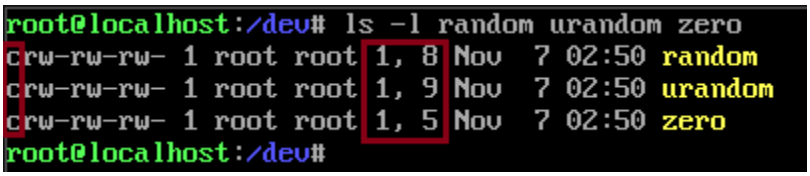
[377] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[378] https://wiki.archlinux.org/title/Device_file
[379] https://medium.com/@boutnaru/the-linux-concept-journey-block-devices-f6f775852091
[380] https://man7.org/linux/man-pages/man1/ls.1.html
[381] https://man7.org/linux/man-pages/man1/mknod.1.html
[382] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[383] https://man7.org/linux/man-pages/man2/mknod.2.html
[384] https://linux.die.net/man/8/lsblk

Character Device File

As we know the philosophy of Linux is that “Everything is a file”. However, not all files are created equally. A block device file is one of the supported file types by Linux[385]. The goal of a “Character Device File” is to provide access for slow devices[386].

Overall, character device files are flagged with “c” in the output of “ls -l”[387]. By default we can find them in the “/dev” directory. Examples of such character device files are: keyboards, serial ports, sound cards and joysticks[388]. Also, the source devices zero, random and urandom[389] are exposed using character device files -as shown in the screenshot below.

Lastly, we can create a block device file using the “mknod” command[390], when using the command we need to provide the major and minor numbers of the device[391] - as shown in the screenshot below. By the way, we can call the “mknod”/”mknodat” syscall[392] in order to create a character device file.



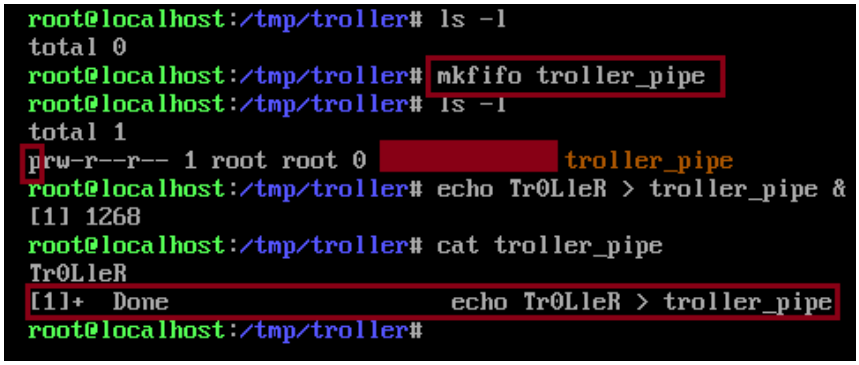
[385] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[386] https://medium.com/@boutnaru/the-linux-concept-journey-character-devices-0c75aa70ceb2
[387] https://man7.org/linux/man-pages/man1/ls.1.html
[388] https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html
[389] https://linux.die.net/man/4/urandom
[390] https://man7.org/linux/man-pages/man1/mknod.1.html
[391] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[392] https://man7.org/linux/man-pages/man2/mknod.2.html

Pipe File (aka Named Pipe/FIFO)

As we know the philosophy of Linux is that “Everything is a file”. However, not all files are created equally[393]. Among them we have a “pipe” file type (aka “FIFO”), which is a way for two completely unrelated processes to communicate between each other. The content of the pipe stays in memory until another connection which reads[394] – as shown in the screenshot below.

Overall, we can create a FIFO (“named pipe”) using the “mkfifo” command[395] or by leveraging the “mkfifo” library function[396] - as shown in the screenshot below.

Lastly, while using “ls -l”[397] in order to list files in a directory, the pipe files are marked with “p” in the output of the command - as shown in the screenshot below. By the way, it is called “FIFO” because the first byte written is the first byte which is read.

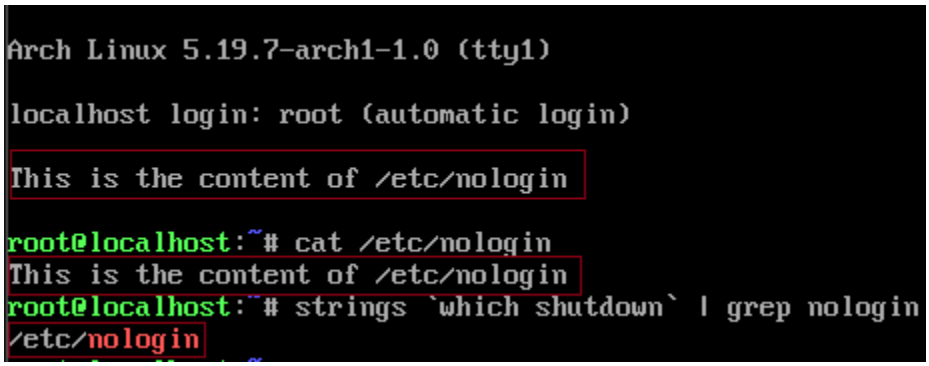


[393] https://medium.com/@boutnaru/the-linux-concept-journey-linux-file-types-4cb622887331
[394] https://www.scaler.com/topics/linux-named-pipe/
[395] https://man7.org/linux/man-pages/man1/mkfifo.1.html
[396] https://man7.org/linux/man-pages/man3/mkfifo.3.html
[397] https://man7.org/linux/man-pages/man1/ls.1.html

/etc/nologin

The goal of the "/etc/nologin" file is to disallow users from logging to a system and notify it will be unavailable for an extended period of time because of a system shutdown or routine maintenance. Thus, in case a user tries to login the content of "/etc/nologin" is displayed to the user and the login process is terminated. By the way, root/superuser logins are not affected[398].

Overall, the "/etc/nologin" file is used by some versions of the "login" utility[399], if it exists the login of non-root users is prohibited[400]. Lastly, in case of a root user the login is performed but the content of the file is still displayed by the "login" utility - as shown in the screenshot below (taken from https://copy.sh/v86/?profile=archlinux). By the way, the "shutdown" utility[401] also creates the "/etc/nologin" file[402].



[398] https://docs.oracle.com/cd/E19683-01/806-4078/6jd6cjs3v/index.html
[399] https://medium.com/@boutnaru/the-linux-process-journey-login-02b6d83ab6c5
[400] https://github.com/mirror/busybox/blob/master/loginutils/login.c#L39
[401] https://linux.die.net/man/8/shutdown
[402] https://unix.stackexchange.com/questions/17906/can-i-allow-a-non-root-user-to-log-in-when-etc-nologin-exists

/proc/kcore (Kernel ELF Core Dumper)

Basically, "/proc/kcore" is a file which is part of the "/proc" pseudo file-system which is used for process information\system information\sysctl[403]. This fire represents the physical memory of the entire system. The total length of the file is the size of the physical memory the system has plus 4 KiB[404]. Thus, using "/proc/kcore" and an unstripped binary of the kernel (/usr/src/linux/vmlinux) we can leverage GDB to check the current state of the kernel.

Moreover, the "/proc/kcore" file is an ELF core dump file - as shown in the screenshot below (taken from https://copy.sh/v86/?profile=archlinux). However, as opposed to generic core dump which captures a single process "/proc/kcore" provides a real-time view of the system as a whole. In case a read of the ELF header/program headers/ELF note is performed they are generated on the fly and sent to the user[405] - as shown in the screenshot below.

Lastly, the implementation of "/proc/kcore" can be found as part of "/fs/proc/kcore.c" in the Linux source code[406]. Each time the a read access is performed on "kcore" the "read_kcore_iter" function is called[407]. By the way, until kernel version 6.3.13 the "read_kcore" function was used instead[408].

```
root@localhost:~# file /proc/kcore
/proc/kcore: ELF 32-bit LSB core file, Intel 80386, version 1 (SYSV), SVR4-style, from 'rw apm=off vga=0x344 video=vesafb:ypan,v
remap:8 root=host9p rootfstype=9p rootf'
root@localhost:~# readelf -h /proc/kcore
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:                              CORE (Core file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0
root@localhost:~#
```

[403] https://man7.org/linux/man-pages/man5/proc.5.html
[404] https://man7.org/linux/man-pages/man5/proc_kcore.5.html
[405] https://schlafwandler.github.io/posts/dumping-/proc/kcore/
[406] https://elixir.bootlin.com/linux/v6.10/source/fs/proc/kcore.c
[407] https://elixir.bootlin.com/linux/v6.10/source/fs/proc/kcore.c#L310
[408] https://elixir.bootlin.com/linux/v6.3.13/source/fs/proc/kcore.c

Mem Device (/dev/mem)

In general, "/dev/mem" is a character device[409] that we can use for reading/patching the memory of the current system. Every byte access references a physical memory address - as shown in the screenshot below. Thus, non-existent locations lead to errors. By default "/dev/mem" has a major number "1" and a minor number of "1"[410]. We can use the "devmem2" utility for reading/writing form/to any location in memory[411]. For supporting "/dev/mem" we need to set "DEVMEM=y" as part of the kernel configuration before compiling it[412].

Moreover, in order to limit "/dev/mem" we need to set the CONFIG_STRICT_DEVMEM variable to "Y". In case it is set to "N" user-processes with root access can access all memory of the system (user-mode/kernel-mode). In case it is enabled and "IO_STRICT_DEVMEM=n" "/dev/mem" allows user-space code to PCI space and BIOS code and data regions, which are needed for DOS emulation and X[413]. We can check the source code of "/dev/mem" as part of the Linux source code at "/driver/char/mem.c"[414].

Lastly, if it is set to "IO_STRICT_DEVMEM=y" user-space access can only idle IO-memory ranges (/proc/iomem). Due to that it can break traditional users of "/dev/mem" like legacy X, DOS emulation and more[415]. We can think about "/dev/mem" as "/dev/kmem" as equal except that the second uses virtual memory addresses while the first uses physical memory addresses.

[409] https://medium.com/@boutnaru/the-linux-concept-journey-character-devices-0c75aa70ceb2
[410] https://man7.org/linux/man-pages/man4/kmem.4.html
[411] https://manpages.ubuntu.com/manpages/focal/en/man1/devmem2.1.html
[412] https://elixir.bootlin.com/linux/v6.11/source/drivers/char/Kconfig#L310
[413] https://cateee.net/lkddb/web-lkddb/STRICT_DEVMEM.html
[414] https://elixir.bootlin.com/linux/v6.11/source/drivers/char/mem.c
[415] https://cateee.net/lkddb/web-lkddb/IO_STRICT_DEVMEM.html

Kmem Device (/dev/kmem)

As with the device file "/dev/mem"[416] also "/dev/kmem" can be used in order to read/write to the main memory of the system. The difference is that "/dev/kmem" performs operations on the kernel virtual memory address space as opposed to "/dev/mem" which does that on the physical address space of the kernel[417].

Moreover, by default "/dev/mem" has a major number "1" and a minor number of "2"[418]. Also, based on the Linux kernel documentation "/dev/kmem" is obsolete[419] and has been replaced by "/proc/kcore"[420].

Lastly, until kernel version "5.12.19" we could set "DEVKMEM=y" while compiling the kernel in order to add the support for the virtual device "/dev/kmem"[421] - as shown below. Thus, the code which creates the device and sets the relevant file operations[422] is compiled as part of the kernel (and later executed when the kernel is loaded). Since kernel version "5.13" it is not supported anymore[423].

```
#ifdef CONFIG_DEVKMEM
        [2] = { "kmem", 0, &kmem_fops, FMODE_UNSIGNED_OFFSET },
#endif
```

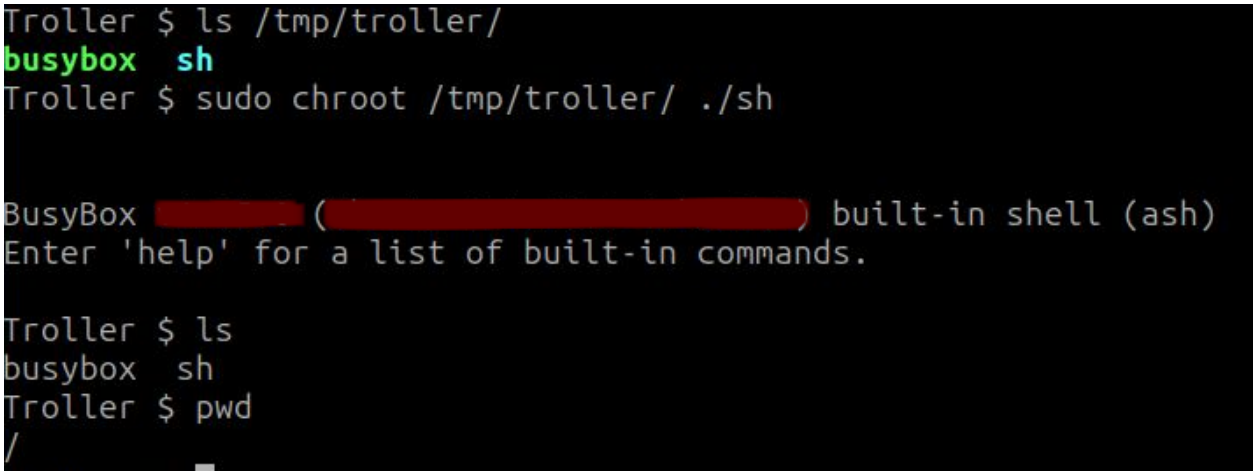
[416] https://medium.com/@boutnaru/the-linux-concept-journey-dev-mem-ee697b16ed3d
[417] https://linux.die.net/man/4/kmem
[418] https://medium.com/@boutnaru/the-linux-concept-journey-major-minor-numbers-56abe372482e
[419] https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/devices.txt
[420] https://medium.com/@boutnaru/the-linux-concept-journey-proc-kcore-kernel-elf-core-dumper-0a1f90d1ad99
[421] https://elixir.bootlin.com/linux/v5.12.19/source/drivers/char/Kconfig#L337
[422] https://elixir.bootlin.com/linux/v5.12.19/source/drivers/char/mem.c#L929
[423] https://elixir.bootlin.com/linux/v5.13/K/ident/CONFIG_DEVKMEM

chroot (Change Root Directory)

chroot is a Linux system call which allows changing the root directory of a calling process to a specific path. After doing so the directory will be used for the path names beginning with "/". The changed root directory is inherited to all children of the calling process. By the way, only privileged processes can call "chroot" - root or with "CAP_SYS_CHROOT" in its user namespace[424].

Moreover, there are different use case (which are not joust security related) for using chroot like: rebuilding initramfs image, reinstalling a bootloader, upgrading/downgrading a package and more[425]. By the way, we can use the "chroot" CLI tool (and not the system call) for preventing access outside the new root directory[426] - as shown in the screenshot below. It is recommended to go over the implementation of the "chroot" syscall[427].

Lastly, we can think about "chroot" as a mitigation/hardening feature (and not a security feature) due to the fact there are specific ways to bypass it[428]. We can find it in use when creating sandboxed environments[429] - they are better solutions than just using "chroot" as described in future writeups (namespaces and seccomp as an example). An example for that is "wu-ftpd" which can run in a chrooted environment for anonymous users[430].



[424] https://man7.org/linux/man-pages/man2/chroot.2.html
[425] https://wiki.archlinux.org/title/chroot
[426] https://linux.die.net/man/1/chroot
[427] https://elixir.bootlin.com/linux/v6.5.5/source/fs/open.c#L593
[428] https://www.redhat.com/en/blog/chroot-security-feature
[429] https://www.lenovo.com/us/en/glossary/what-is-chroot/
[430] https://www.ariadne.ac.uk/issue/20/unix/

Namespaces

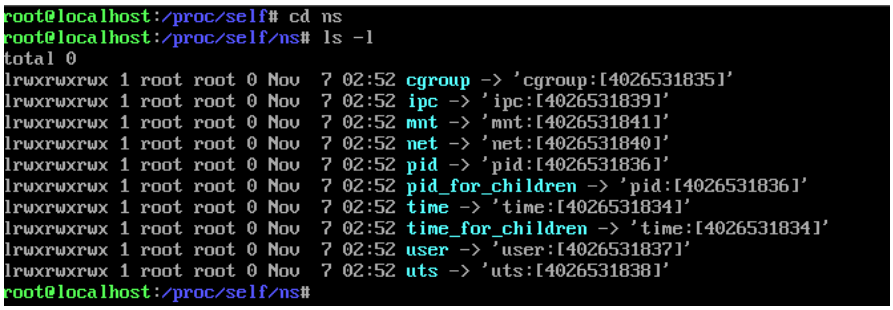
The goal of namespaces is to create an illusion that resources seen by a process are the only resources the system has. Basically we can think about namespace as an isolation technology for processes which are sharing the same kernel (versus processes running with different kernels such as two VMs on the same hypervisor). Namespaces were first introduced as part of kernel 2.4.19. By the way, namespaces are one of the basic building blocks of containers.

Thus, if we have two different processes (that are part of different namespaces) a change made by one of them on a resource (that is part of one of the namespaces) is not visible to the second process (unless there is a bug/security vulnerability). More information about namespace can be found using "man 7 namespaces"[431].

Moreover, in order to use namespaces we can leverage different syscalls: clone, setns, unshare and ioctl_ns. When using "clone" ("man 2 clone") to create new processes we can pass different flags that will create new namespaces that we want the new process to be part of. With "setns" ("man 2 setns") we can add a process to an existing namespace. By using "unshre" ("man 2 unshare" we can also move a process to a new namespace (it is different from clone because it is not before the creation of the process). Finally, with "ioctl_ns" ("man 2 ioctl_ns") to perform operations such as discovery regarding namespaces.

It is important to know that, today there are 8 different types of namespaces: User, PID, UTS (hostname), Time, IPC, Network, Cgroup and Mount. I am going to elaborate on each and one of them in a separate writeup. Also, there is a suggestion to add a syslog[432] namespace which was not merged to the kernel until now.

We can use "/proc" ("man 5 proc") to query information about namespaces of the running process. The information about namespaces is stored under "/proc/[pid]/ns" - as shown in the screenshot below (taken using copy.sh). Every link shown in the screenshot corresponds to a namespace. The two links ending with "for_children" represent the information about the namespaces of the child processes created by the process.



[431] https://man7.org/linux/man-pages/man7/namespaces.7.html
[432] https://lwn.net/Articles/562389

PID namespace

The goal of PID namespaces is to isolate the "Process ID number space". Thus, different processes in distinct PID namespaces can have the same PID. When a new PID namespace is started the first process gets PID 1 (so we don't have a new swapper[433]). In order to use PID namespaces we have to ensure that our kernel was compiled with "CONFIG_PID_NS" enabled[434].

Moreover, PID namespace can also be nested , since kernel 3.7 the maximum nesting depth is 32. A process is visible to every other process in the same PID namespace or any direct ancestor PID namespace. The opposite way does not work, a process in a child PID namespace can't see a process in a parent PID namespace[435].

Also, "/proc" will show only processes which are visible in the PID namespace of the process that executed the "mount" operation for "/proc"[436]. If we want to see the number of the last pid that was allocated in our PID namespace we can use "/proc/sys/kernel/ns_last_pid"[437] - as you can see in the screenshot below.

Lastly, a nice fact to know is that when we pass a pid over a unix domain socket to a process which belongs to another PID namespace, it is resolved to the correct value in the receiving process' PID namespace[438].

[433] https://medium.com/@boutnaru/the-linux-process-journey-pid-0-swapper-7868d1131316
[434] https://man7.org/linux/man-pages/man7/pid_namespaces.7.html
[435] https://www.schutzwerk.com/en/blog/linux-container-namespaces03-pid-net/
[436] https://lwn.net/Articles/531419/
[437] https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#ns-last-pid
[438] https://man7.org/linux/man-pages/man7/pid_namespaces.7.html

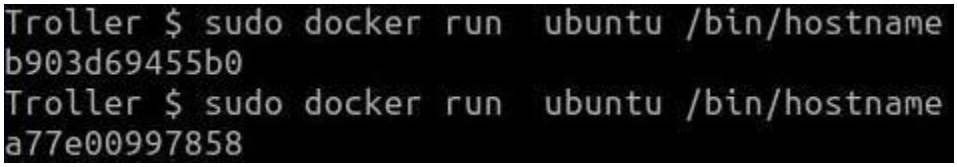
UTS namespace

In the first part of the series we have talked generally about what namespaces are and what we can do with them. Now we are going to deep dive to the different namespaces starting with UTS (Unix Time Sharing).

By using UTS namespaces we can separate/isolate/segregate the hostname and the NIS (Network Information Service) domain of a Linux system. Just to clarify, NIS is a directory service (it has some similarities to Microsoft's Active Directory) that was created by Sun and later was discontinued by Oracle . Thus, UTS today is focused mainly on separating hostname.

In order to get the information described above we can use the following syscalls: uname ("man 2 uname"), gethostname ("man 2 gethostname") and getdomainname ("man 2 getdomainname") - we also have a parallel set syscall for each one of them. For supporting UTS namespaces the kernel should be compiled with "CONFIG_UTS_NS"[439].

Container engines (such as docker) use different namespaces to isolate between different containers, even if they were created from the same image - as shown in the screenshot below.



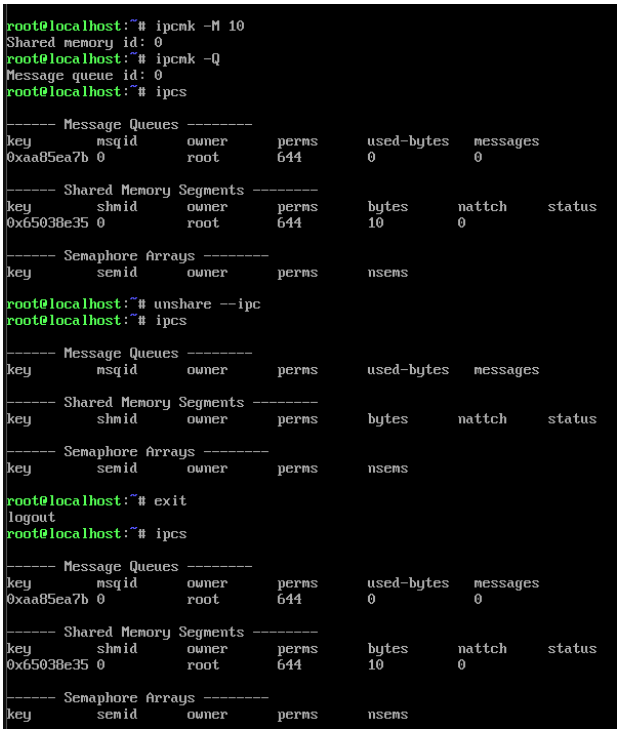
[439] https://github.com/torvalds/linux/blob/master/include/linux/utsname.h#L32

IPC namespace

The goal of the IPC namespace is to isolate between different IPC resources like message queues, semaphores and shared memory. We are talking both on System V IPC objects[440] and POSIX message queues[441]. In order to use “IPC namespaces” the kernel should be compiled with CONFIG_IPC_NS enabled[442].

We can use “/proc” in order to retrieve information about the different IPC objects in an “IPC namespace”. Regarding POSIX message queues we have “/proc/sys/fs/mqueue”. In case of System V IPC objects we have “/proc/sysvipc” and specific file in “/proc/sys/kernel” (msgmax, msgmnb, msgmni, sem, shmall, shmmax, shmmni, and shm_rmid_forced). For more information I suggest reading proc’s man page[443].

Lastly, all IPC objects created in an “IPC namespace” are visible only to all processes/tasks that are members of the same namespace - as shown in the screenshot below. In the demonstration the namespace was created using “unshare”[444], the IPC resource was created using “ipcmk”[445] and the show information about System V IPC resources using “ipcs”[446].



[440] https://man7.org/linux/man-pages/man7/sysvipc.7.html
[441] https://man7.org/linux/man-pages/man7/mq_overview.7.html
[442] https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html
[443] https://man7.org/linux/man-pages/man5/proc.5.html
[444] https://man7.org/linux/man-pages/man1/unshare.1.html
[445] https://man7.org/linux/man-pages/man1/ipcmk.1.html
[446] https://man7.org/linux/man-pages/man1/ipcs.1.html

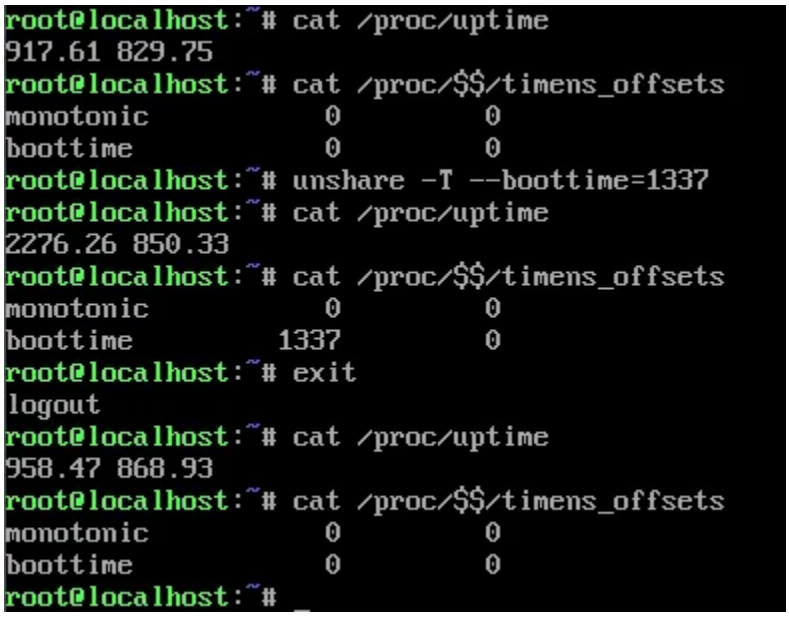
Time namespace

By using time namespaces we can separate/isolate/segregate and thus virtualize the values of two system clocks[447]. We are talking about “CLOCK_MONOTONIC” and “CLOCK_BOOTTIME”.

“CLOCK_MONOTONIC” goal is to represent an absolute elapsed wall-clock time since some arbitrary, fixed point in the past. It is important to understand that it isn't affected by changes in the system time-of-day clock. As opposed to “CLOCK_REALTIME” which can change based on configurations/NTP (Network Time Protocol) data[448]. Moreover, “CLOCK_MONOTONIC” does not measure time spent during suspend. If we want a monotonic clock that is running during suspend we need to use “CLOCK_BOOTTIME”[449].

Thus, all the processes in the same time namespace share the same values of the clocks explained above. Due to that, it affect the results of the following syscalls: timer_settime (“man 2 timer_settime”, clock_gettime (“man 2 clock_gettime”), clock_nanosleep (“man 2 clock_nanosleep”), timerfd_settime (“man 2 timerfd_settime”). Also, the content returned from “/proc/uptime” is affected.

In order to create a new time namespace we have to use the unshare syscall (“man 2 unshare”) and pass the “CLONE_NEWTIME” flag. You can see an example of that using the “unshare” cli tool (“man 1 unshare”) in the screenshot below. In order to see the difference between the initial time namespace and the process’ time namespace we can use “/proc/[PID]/timens_offsets” also shown in the screenshot below.



[447] https://man7.org/linux/man-pages/man7/time_namespaces.7.html
[448] https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic
[449] https://linux.die.net/man/2/clock_gettime

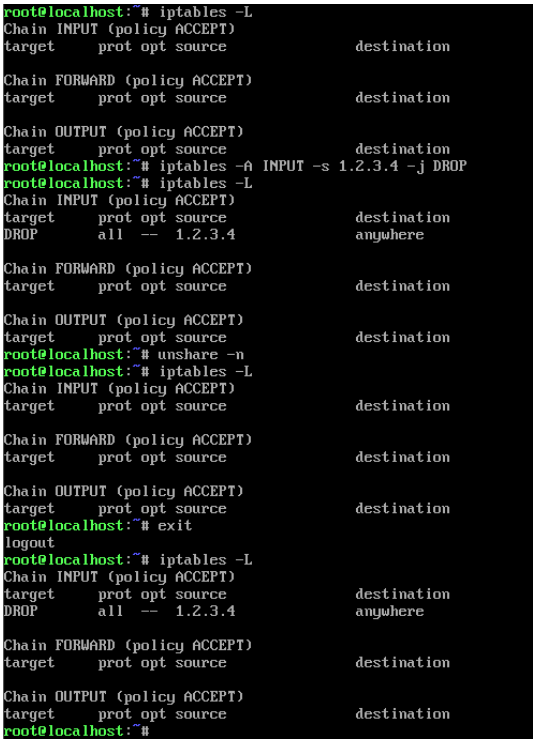
Network namespace

First, in order for the kernel to support network namespaces we need to compile the kernel with "CONFIG_NET_NS" enabled. Overall, network namespaces can separate/isolate/segregate between the different system resources which are associated with networking under Linux. Among those resources are: firewall rules, routing tables (IP), IPv4 and IPv6 protocol stacks, sockets, different directories related to the networking subsystem (like: "/proc/[PID]/net", "/proc/sys/net", "/sys/class/net" and more), etc[450].

By the way, unix domain sockets are also isolated using network namespaces ("man 7 unix"). It is important to understand that a physical network device can exist in one network namespace at a time (singleton). In case the last process in a network namespace returns/exits, Linux frees the namespace which moves the physical network device to the initial network namespace.

Moreover, in case we want to create a bridge to a network device which is part of a different namespace we can use a virtual network device. It can create tunnels between network namespaces[451].

Lastly, you can see an example of creating a network namespace in the screenshot below. As you can see an iptables rule is created but it is not relevant to the newly created network namespace.

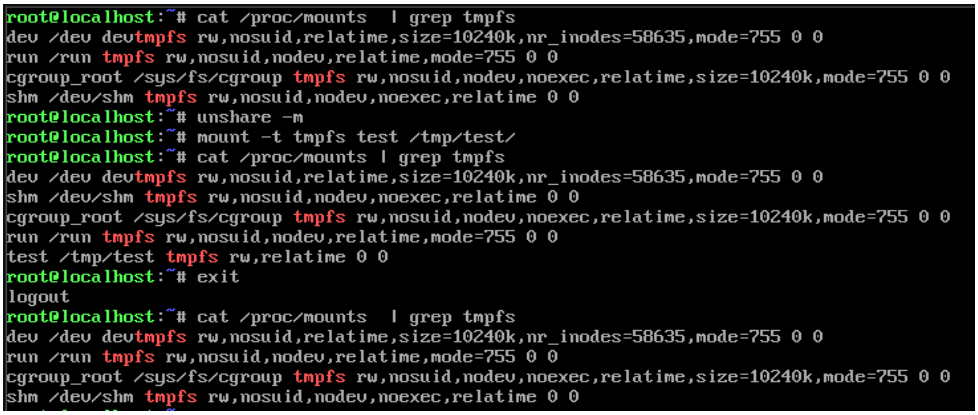


[450] https://man7.org/linux/man-pages/man7/network_namespaces.7.html
[451] https://man7.org/linux/man-pages/man4/veth.4.html

Mount Namespace

The goal of mount namespaces is to provide isolation regarding the list of mounts as seen by the process/tasks in each namespace. By doing so different processes/tasks that belong to different mount namespaces will see distinct directories hierarchies[452].

Using "/proc" we can inspect the mounting points visible for specific processes using "/proc/[PID]/mounts, "/proc/[PID]/mountinfo" and "/proc/[PID]/mountstats"[453]. You can see the different outputs when reading the data in different mount namespace - as seen in the screenshot below.



After the implementation of mount namespaces the isolation they have created a problem. Think about a case when we add some device that we want to be visible on every namespace, for that we need to execute a "mount" command on each namespace. To overcome this the shared subtree feature was introduced in kernel 2.6.15[454].

By using shared subtrees we can propagate mount/unmount events between distinct mount namespaces[455]. It is designed to work between mounts that are members of the same peer group. Thus, "peer group" is defined as a group of vfsmounts that propagate events between each other[456]. Also, we can control the propagation using the mount system call by passing one of the following "mountflags"[457]: MS_SHARED, MS_PRIVATE, MS_SLAVE, or MS_UNBINDABLE[458]. For more information about shared subtrees I suggest reading the kernel documentation[459].

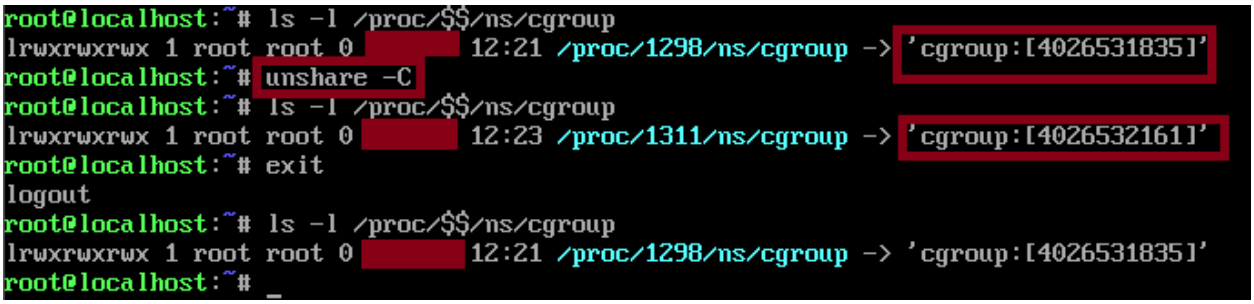
[452] https://man7.org/linux/man-pages/man7/mount_namespaces.7.html
[453] https://man7.org/linux/man-pages/man5/proc.5.html
[454] https://lwn.net/Articles/689856/
[455] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/sect-using_the_mount_command-mounting-bind
[456] https://www.redhat.com/sysadmin/mount-namespaces
[457] https://hechao.li/2020/06/09/Mini-Container-Series-Part-1-Filesystem-Isolation/
[458] https://man7.org/linux/man-pages/man2/mount.2.html
[459] https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt

cgroup namespace

As a reminder, “cgroups” is a feature of the Linux kernel used for limiting system resources (CPU/Memory/IO) of specific processes organized in hierarchical groups[460]. For creating a new “cgroup namespace” we can use the “CLONE_NEWCGROUP” flag while calling the “clone”[461] syscall. It is used for isolating the “cgroup” root directory[462].

Overall, “cgroup namespaces” virtualize the view of a process's cgroups. We can also get information about the cgroup namespace per process/task from the proc filesystem: “/proc/<PID>/cgroup” and “/proc/<PID>/mountinfo”. In order to use cgroup namespaces the kernel should be compiled with “CONFIG_CGROUPS” enabled[463].

Lastly, the creation of a new cgroup namespace can be done using the “clone”[464] syscall and\or the “unshare”[465] syscall by providing the “CLONE_NEWCGROUP” flag (since Linux 4.6). For doing so we need the “CAP_SYS_ADMIN” capability[466]. This can be done also with the “unshare” utility[467] by leveraging the “-C” switch - as shown in the screenshot below (taken using https://copy.sh/v86/?profile=archlinux).



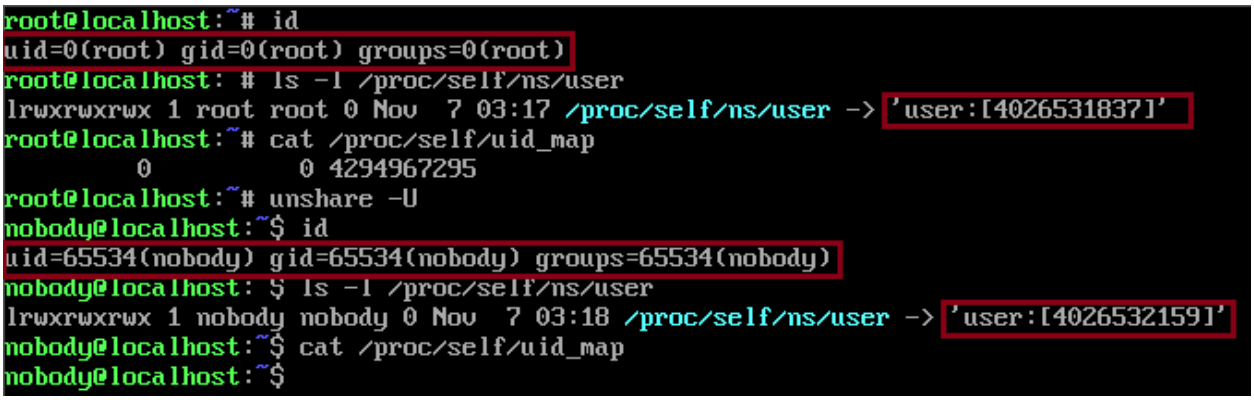
[460] https://medium.com/@boutnaru/linux-cgroups-control-groups-part-1-358c636ffde0
[461] https://man7.org/linux/man-pages/man2/clone.2.html
[462] https://man7.org/linux/man-pages/man7/namespaces.7.html
[463] https://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html
[464] https://man7.org/linux/man-pages/man2/clone.2.html
[465] https://man7.org/linux/man-pages/man2/unshare.2.html
[466] https://man7.org/linux/man-pages/man7/capabilities.7.html
[467] https://man7.org/linux/man-pages/man1/unshare.1.html

User Namespace

In general, “User Namespace” is a feature of the Linux kernel which allows isolation of group/user identifiers mapping. By using this feature every user namespace has its own set of group/user identifiers. This means that processes/threads (tasks in Linux's lingo) executing in different namespaces can have different privileges/permissions even if their user/group ID is the same number. Due to that, this feature is super relevant for container environments[468].

Moreover, User namespace is used for isolating identifiers and attributes which are security-related. Among those are: user/group IDs, root directory, keyring[469] and capabilities[470]. Thus, a process can be privileged inside a user namespace but unprivileged outside the namespace[471]. We can check out the user namespace of a specific process using “/proc/[PID]/ns/user” and also the uid/gid maps using “/proc/[PID]/uid_map” or “/proc/[PID]/gid_map”[472] - an example of usage is shown in the screenshot below (taken from copy.sh).

Lastly, we can use the “unshare” system call[473] or the “unshare” utility[474] for running processes in a new user namespace. In the screenshot below we can see an example of that by using the “unshare” utility. Also, we can leverage the “clone”[475] or “setns”[476] syscalls for creating/joining a user namespace. We can also check the source code of the Linux kernel which implements user namespaces[477].



[468] https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-security/namespaces/user-namespace
[469] https://medium.com/@boutnaru/linux-keyrings-d4ad07c091b3
[470] https://medium.com/@boutnaru/linux-security-capabilities-part-1-63c6d2ceb8bf
[471] https://man7.org/linux/man-pages/man7/user_namespaces.7.html
[472] https://man7.org/linux/man-pages/man5/proc.5.html
[473] https://man7.org/linux/man-pages/man2/unshare.2.html
[474] https://man7.org/linux/man-pages/man1/unshare.1.html
[475] https://man7.org/linux/man-pages/man2/clone.2.html
[476] https://man7.org/linux/man-pages/man2/setns.2.html
[477] https://elixir.bootlin.com/linux/v6.9/source/kernel/user_namespace.c