Linux Kernel for Newbies

Kaloyan Krastev*

Thursday 9th November, 2023 revision 2893

*Triple Helix Consulting

table of contents

1 introduction

The definitions are required to understand the relation between the kernel, the operating system and the user applications. These occupy distinct parts of the memory called kernel-space and user-space. The kernel, multitasking, preemption, symmetrical multiprocessor support and kernel modules are defined in Section 2.

Linux and some important kernel features are covered in Section 3. In this paper, Linux always designates the kernel while the corresponding operating system is referred as **GNU!** (**GNU!**)/Linux. That is why Linux should not be compared to operating systems but their kernels like the original Unix kernel, the BSD kernel, NT, and XNU, the last two designed by Microsoft Corporation and Apple Inc.

Finally, limited technical details about Linux images and kernel releases are available in Section 4.

2 definitions

An **OS!** (**OS!**) is a **SW!** (**SW!**) environment for running applications on computing devices. In other words, the **OS!** provides user applications with a standard interface to system **HW!** (**HW!**) and **SW!** resources. The main components required for the functioning of **OS!**s follow. Strictly speaking, the last two items are not considered as integral parts of the **OS!**. These are commonly defined as **OS!** users.

- bootloader => take care of the boot sequence
- kernel => manage **HW!** and **SW!** resources
- shell => command-line interface to interact with the **OS!**
- services => provide functionalities to applications
- graphical server => display graphical content
- applications => perform different user tasks

2.1 kernel

The kernel[2] is a critical component of an **OS!** responsible for low-level resource management and communication with **HW!**. Famous examples are NT (New Technology) kernel designed by Microsoft Corporation and XNU (X is Not Unix). The last one is an Unix-like kernel from Apple Inc.

2.2 multitasking

Multitasking is about running processes. It is important to understand that a single processor may run only one process at a time.

Even on multiprocessor platforms, the number of processes exceeds the number of processors by at least two to three orders of magnitude. Fortunately, most **OS!**es are able to interleave execution of multiple processes.

In *cooperative multitasking*, a process does not stop running before it has decided itself, in example, when the task is completed. I wonder why it is called like that because there is nothing *cooperative* here. In this approach, a single process could monopolize the machine processor time and potentially bring the entire **OS!** down.

Fortunately, modern **OS!**es employ another solution. In this approach, called *preemptive multitasking*, the kernel grants processes tiny pieces of processor time called *time-slices*. Their duration is calculated by dedicated algorithms. When a time-slice is over, the kernel interrupts the corresponding process in a task called *preemption*, saving the entire working context in order to resume this process later, when the next time-slice will be granted to it. With a typical time-slice length on the order of *milliseconds*, the preemption gives human beings the attractive illusion of many processes running simultaneously. Unfortunately this is not the case.

Symmetrical multiprocessor (SMP) support is a common **HW!** feature supported by the modern **OS!**es. It allows for more efficient management and usage of multiple processors. This is achieved by considering all processors identical. In addition, they may share **HW!** resources like memory and various devices between them. The purpose of SMP support is to ensure that multiple processors work together in harmony to deliver improved performance.

2.3 kernel modules

Linux supports dynamic insertion and removal of code at runtime. This code is grouped in loadable binary objects called modules. Module support allows **OS!**es to have a compact base kernel image without missing optional features and drivers supplied via modules.

A device driver, in example, could be compiled as a part of the kernel or separately - as a kernel module. This depends on the kernel build configuration. As a part of the kernel, the driver is loaded on boot and unloaded on shut down. As a module it could be loaded or unloaded anytime the system may need so. The last solution decreases kernel size and Linux boot time but may potentially decrease **OS!** performance.

3 linux

The initial version of Unix was created in Bell Labs in 1969. The design principles like hierarchical file system, command-line interface and multi-user support have been followed in many modern **OS!**es. However, it was a commercial product with a proprietary development model and practically no development community.

Linux is an **OS!** kernel that was originally developed by Linus Torvalds twenty years later in 1991. In short, it is an Unix-like kernel, but it has a source code freely available to the public. Linux is used as a kernel in multiple **OS!**es, commonly referred to as **GNU!**/Linux distributions.

There are two families of **OS!** kernels - monolithic and micro-kernels. Micro-kernels run as multiple processes usually called servers. As a monolithic kernel, Linux is designed to run as a single piece of **SW!** in a special part of the memory called kernel-space.

In contrast, applications run a virtual environment, provided by Linux, called user-space. The kernel makes each application feel like it has all available resources while, in fact, it shares with the other applications and it has exactly as memory and processor time as Linux has granted. This control makes the main difference with older Unix kernels.

If Linux differs from classic Unix systems, the reason is that Linus Torvalds and his community behind the kernel were willing and able to chose the best solution for any problem. In addition, they have invented new approaches when they were not happy with existing. Readers may find the most important characteristics of the kernel in the following subsection.

3.1 features

These are four characteristics of the kernel that make it unique.

3.1.1 free

Linux is free in all respects by design. Developers implement only features that solve real problems, have a clean design and a simple implementation. In contrast, everybody with a good idea may contribute the open development model.

3.1.2 configurable

Linux is highly configurable. With a proper configuration, the kernel is built and run on desktops, servers, mobile and embedded devices, super-computers and robots. In 2021, there were over two billion devices running Linux. That includes a large number of smartphones running Android, which uses a Linux kernel, and hundreds of millions of set-top boxes, smart TVs, and Wi-Fi routers, not to mention a very diverse range of devices such as vehicle diagnostics, weighing scales, industrial devices, and medical monitoring units that ship in smaller volumes[4].

3.1.3 modular

Although monolithic, Linux supports dynamic loading and unloading of kernel code on demand. Such code is organized in kernel modules. They are stored in $/lib/modules/< kernel\ version >/kernel$. Use $lsmod$ to list active kernel modules. Alternatively dump the file named $/proc/modules$. To get more information of a module use $modinfo$ and if you ever need to load or unload a kernel module, read the documentation of the *modprobe* command-line tool.

3.1.4 preemptible

To be precise, the multitasking algorithm depends on the kernel build configuration. Linux could have a voluntary preemption or no preemption at all. In addition, it could be compiled as a preemptible as well as a fully preemptible kernel. The last option is useful in real-time environments. Commercial Unix **OS!**es like Solaris and IRIX also have preemptive kernels.

3.2 functionality

Linux manages **HW!** resources, provides essential services and enables communication between **SW!** applications and the **HW!**. The following key functions define the purpose of the kernel.

3.2.1 process scheduler

Processes executing in user-space may be running or waiting for a time-slice to run. In both cases their process state is TASK_RUNNING. The process scheduler decides which process to run, when, and for how long. It divides the processor time between the runnable processes. Usually these are more that available processors and while some processes are running the others are waiting to run. The scheduler takes the fundamental decision which process to run next. Completely Fair Scheduler (CFS) is implemented in Linux as a main process scheduler since 2007. To learn more, read about the CFS scheduling algorithm.

3.2.2 memory management

The memory granted to user-space processes by the kernel has a virtual allocation. This way applications work like they have all available memory. In fact, they share RAM with many other processes

and have a restricted access. The MMU (memory management unit) is the **HW!** that manages memory and performs virtual to physical address translation.

3.2.3 system calls

Provide an abstracted **HW!** interface for user-space processes. System calls enable interaction with the kernel. This way applications may request **OS!** resources and services. These are granted by the kernel in a defined and controlled manner. Usually programs wait for the result of a system call to continue execution. Figure 1 is a diagram of one user application *read* system call to the kernel.

Each system call type is assigned a standard number that cannot be modified. Otherwise code written before the modification may not work. System call numbers are defined in the following file.

$/usr/include/asm/unistd\_64.h$

It belongs to the package $linux-api-headers$ required by *glibc*. This is a *c* header with a complete list of three hundred sixty-two system calls.

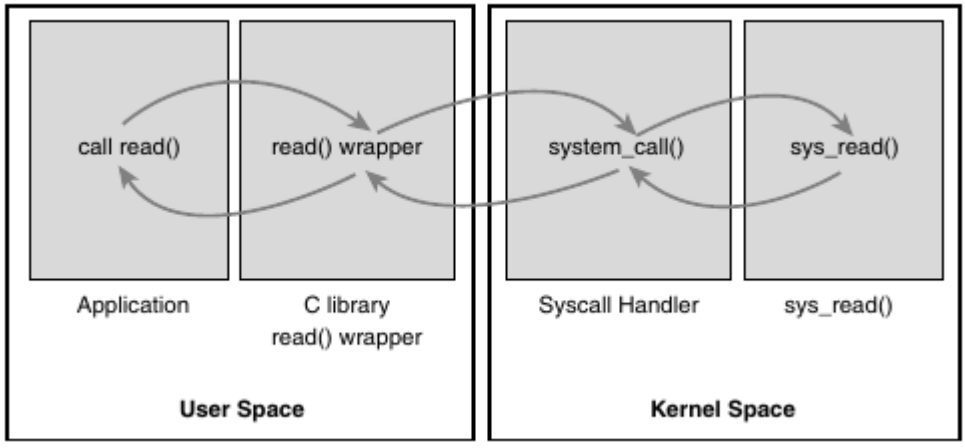

Figure 1: a system call[2]

3.2.4 interrupts

Allow **HW!** devices and **SW!** processes to temporarily suspend the current processor execution and transfer control to a specific routine called *interrupt handler*. Interrupts are asynchronous and may occur any time. They handle **HW!** events that require immediate attention like keyboards, mice, network and storage devices. When an interrupt is requested, the processor saves the interrupted process context in order to resume it after the interrupt handler has completed its tasks.

4 exercises

When compiled, Linux is a single binary file, very often called *image*. It is loaded on boot into a supervisor mode called kernel-space. This space provides unrestricted access to the **HW!**. The image has a size of around 13 MB. In comparison, the source code, written in *c*, takes around 1.5 GB on a storage device. These are mainly **HW!** drivers for different devices and source code for various processor architectures.

4.1 kernel releases

Linux versions contain three numbers divided by dots. The first number is the major version, followed by the minor one and the last correspond tiny improvements like bug fixes and security updates. In example mine is 6.4.12. There is not any specific interval between two releases. Various considerations may cause releases of new versions, but I do not want to discuss this topic here.

Some **GNU!**/Linux distributions suggest automatically kernel updates, some not. In example, openSUSE do that after each stable release. In contrast, on Manjaro Linux[1], you are expected to update the kernel yourself.

4.2 kernel update

There is a dedicated graphical interface on Manjaro[3]. Look for the *Kernel* section of *systemsettings*. Here one may install/uninstall kernel releases of her/his choice, if (s)he has root privileges. Usually, in addition to the active Linux kernel, one or two older versions are kept on the boot partition, where the bootloader has access during the boot sequence. This allows to switch back to an older version, if the new kernel fails to load. To be honest, after more than

twenty years work on various **GNU!**/Linux distributions, this has never happened to me. You may think that I am just lucky but, in fact, before being released, a new kernel is tested and verified with multiple build configurations by many different users on various process architectures. Find available kernel images on your system in `/boot/vmlinuz-[x.y]-[arch]`, where `x` is the major, `y` is the minor kernel version and `arch` is the processor architecture.

4.3 activate kernel images

Linux releases with **LTS!** (**LTS!**) are recommended. Avoid kernels that are not supported by the system. By default, the location of the last kernel image loaded on boot is saved and next time it is loaded automatically. Learn next how to modify this behaviour in the bootloader configuration, in this case - GRUB (GRand Unified Bootloader).

This is how to activate a different kernel release. This should load a specific kernel image during next boot. Logically, one may chose one of the images already installed on the *boot* partition.

- to list GRUB entries, including installed kernel releases, use the following command

  sudo grep 'menuentry ' /boot/grub/grub.cfg | cut -f 2 -d "'" | nl -v 0

  In this list, one positive integer number corresponds each image. Remember the number of the kernel to activate.

- comment the directive GRUB_SAVEDEFAULT in `/etc/default/grub`

- run *sudo grub-reboot* followed by the number in question

On next reboot the corresponding kernel should be loaded.

Readers may have noticed that all commands are executed with root privileges. This is not by chance. According to my experience, if the bootloader has a wrong configuration, the probability of a crash during boot sequence is quite high. Always verify changes in advance and be prepared to revert changes if a failure occurs.

5 outlook

The article covers important Linux features and functions as well as definitions needed to understand them. In addition, readers may learn how to install new kernel releases and how to activate a particular kernel image.

bibliography

[1] Vitor Lopes. *Manjaro*. 2023.
URL: https://manjaro.org (visited on 2023).

[2] Robert Love. *Linux Kernel Development*. Addison-Wesley, 2010.

[3] Atanas Georgiev Rusev. *Manjaro Linux User Guide*.
Packt Publishing Limited, 2023.

[4] Frank Vasquez and Chris Simmonds.
Mastering Embedded Linux Programming.
Packt Publishing Limited, 2021.