K22 Tutorial 2

A top level overview of the Linux kernel

K22 Crew

2026-10-11

dit@UoA

The Linux Kernel

Introduction

The Linux kernel is based on a **monolithic kernel** architecture.

- This means that all code that is part of the kernel runs in **privileged** (supervisor/kernel) mode and has full access to all hardware.

Introduction

The Linux kernel is based on a **monolithic kernel** architecture.

- This means that all code that is part of the kernel runs in **privileged** (supervisor/kernel) mode and has full access to all hardware.

It is also structured as a gigantic mono-repo with all actively maintained and recommended parts of the kernel being part of the main repository.

- Anything that is **not** in the main repository is not endorsed to be used.

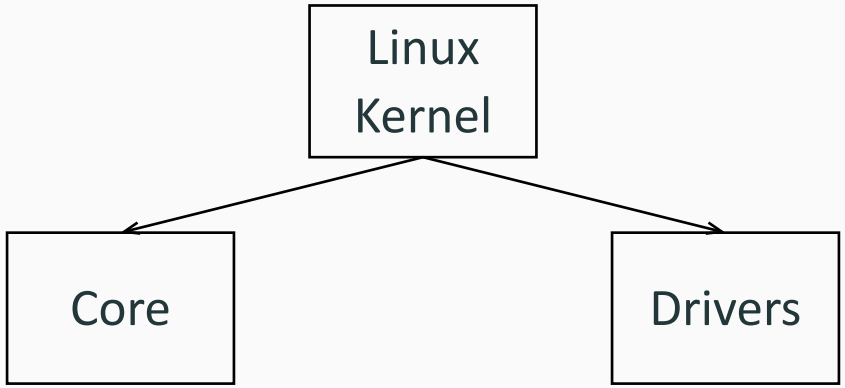
Structure

While the kernel is structured very monolithically in regards to outside access, it is very flexible and modular on the inside.

Structure

While the kernel is structured very monolithically in regards to outside access, it is very flexible and modular on the inside.

The kernel is split up into the **kernel core** and the **kernel drivers**.



We will work on the **Core** of the kernel throughout this semester.

The Core

The Linux kernel core is probably the most important part of the kernel. It is split into what is referred to as **subsystems**. Each subsystem deals with a different part of managing hardware and processes. Examples of subsystems, include:

- `init`: unifies architecture specific boot sequences into a clear control flow
- `kernel`: basic system calls
- `sched`: processor time management and process scheduling
- `mm`: handles everything virtual memory **and** physical memory
- `fs`: VFS (Virtual Filesystem, a kernel level abstraction to unify filesystem implementations) and implementations of actual filesystem code

Drivers

As discussed in the course, a large part of what a kernel does is **managing hardware**. In order for us to realize how big a part that is, here's some statistics:

- Linux Kernel: 30,518,973 lines of code spread over 82,376 files
- 20,186,695 (about 60%) of those is in the `drivers` directory and is spread over 35,816 files

Drivers

As discussed in the course, a large part of what a kernel does is **managing hardware**. In order for us to realize how big a part that is, here's some statistics:

- Linux Kernel: 30,518,973 lines of code spread over 82,376 files
- 20,186,695 (about 60%) of those is in the `drivers` directory and is spread over 35,816 files

It is obvious that most of the kernel is simply just code to teach it how to handle different devices. Why is that? Because if even ONE person has to use that device, they need to write a driver for it. And due to the open source nature of the Linux kernel, they **choose** to push it upstream.

Drivers

As discussed in the course, a large part of what a kernel does is **managing hardware**. In order for us to realize how big a part that is, here's some statistics:

- Linux Kernel: 30,518,973 lines of code spread over 82,376 files
- 20,186,695 (about 60%) of those is in the `drivers` directory and is spread over 35,816 files

It is obvious that most of the kernel is simply just code to teach it how to handle different devices. Why is that? Because if even ONE person has to use that device, they need to write a driver for it. And due to the open source nature of the Linux kernel, they **choose** to push it upstream.

A big part of the popularity of Linux is exactly this. It's really hard to find a device that it **does not** support.

Modularity and config

Due to the size of the kernel and the multitudes of supported features and devices, it is necessary for builds to be able to either disable/enable features or load them **only when needed**. The Kconfig system was designed with that in mind. It is an extensive configuration tool that allows someone to choose **exactly** what they want their kernel to support.

Furthermore, to avoid loading all driver and feature code ad-hoc, the module system has been implemented. Typically, one would include their driver and feature code, directly into their kernel binary. The module system, allows people to choose not to do that and instead, keep a packaged up version of a feature/driver called a **Module** on their machine and enable it only when it is required. If you recall, we used this when setting up KVM, which is an optional feature of the kernel that is added in the form of a module.

System Call Implementation

API

In this section we will design and implement a system call in the Linux kernel.

API

In this section we will design and implement a system call in the Linux kernel. We choose the following API for the system call:

```
/**
* k22_hello - Say Hello to userspace
*
* Appends the string "Hello " at the beginning of the null-terminated string provided in @input,
which as a length of @in_size.
* Writes the null-terminated result in the @output buffer, which has at most length @out_size.
*
* Also prints a message to the kernel log.
*
* Returns: 0 on success, -ERROR if something went wrong
*/
int k22_hello(const char __user *input, size_t in_size, char __user *output, size_t out_size);
```