Abstract

Every computer science enthusiast is proficient in their operating system of choice. They may even know its underlying working from an old operating system course they took in college. However, it is usually the case that their knowledge and understanding is limited to theory and writing low-level system code is often considered an insurmountable challenge.

This project hopes to change this attitude by developing a minimal yet functional 32-bit operating system that can be used in conjunction with theoretical teaching to promote and introduce systems programming. A minimal kernel guarantees easier to read source code (as opposed to the 27 million SLOC Linux kernel) and provides a gentler introduction to kernel development.

The kernel will include a full keyboard and mouse driver and will have support for VGA text-mode and graphics. It will also contain a limited libc implementation with a streamlined build process.
Additionally, this project will serve as an illustration for good development practices (code reuse, clean architecture, unit testing).

The 32-bit kernel will be written in C (and possibly some C++) with a little of assembly for the truly low-level aspects. A modern language such as Rust was deliberately not chosen because they often hide certain implementation details that would potentially lead to gaps in knowledge.

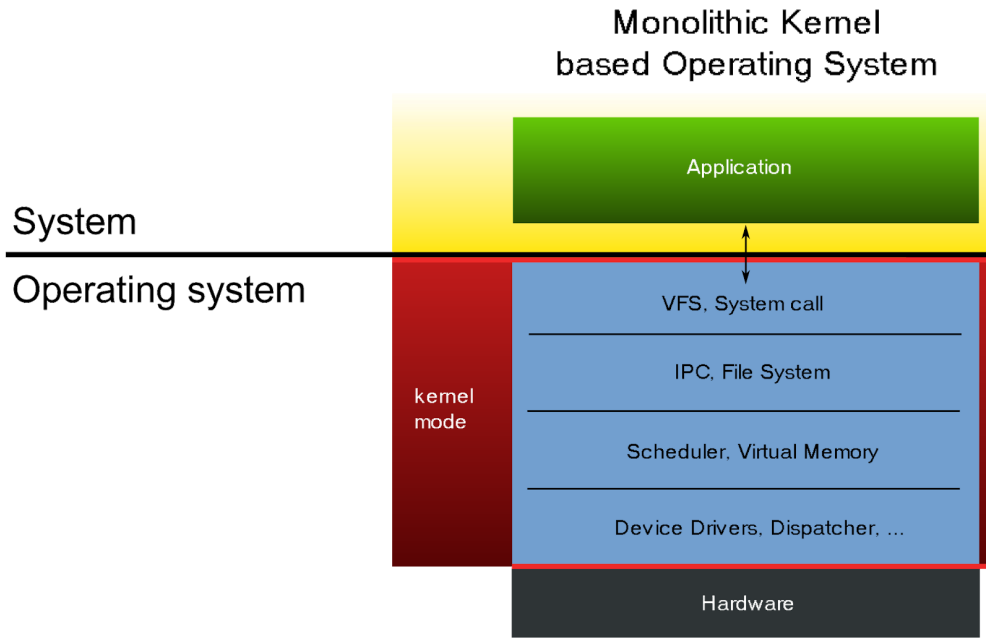
Kernel architecture design

There are three types of kernel architectures which are

-Micro Kernel

- Monolithic Kernel

- Hybrid Kernel



The classification is based on what is in kernel space and what is in user space. There are four protection rings that represent the access a piece of software has to the hardware and system: Out of these only two rings are of interest which are:

1. Ring 0

Code running in Ring 0 is said to be in supervisor mode. It has complete access to the hardware and system.

2. Ring 3

Code running in Ring 3 is said to be in user space. It has no direct access to hardware or the system.

Instead it accesses the system through system calls.

These protections are not simply implemented by the Operating system. Instead they are a part of the CPU architecture. They are activated by asm code.

In a micro kernel very limited amount of code is is in kernel space. For example, most drivers are in userspace.

In a monolithic kernel device drivers and other similar modules operate in kernel space.

A hybrid kernel is somewhere in the middle of what runs in kernel space and user space.

This OS has a monolithic design. Its is motivated by two facts which are:

1. The Linux kernel is monolithic. So if we use a micro kernel architecture, students will not be able to transfer the knowledge that they acquired from our project into linux development space.

2. Our kernel operates completely in ring-0 i.e without any restrictions. Implementing user-mode is unnecessary since we have no user-mode programs. Additionally students need to be in ring-0 to understand how the hardware works.

Build process

The build architecture is a little involved because we are using the C programming language which does not have any real tooling for package management. Instead compiling and linking each translation unit must be done manually.

A tool called GNU Make is used to manage the compilation process. GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. It does this by executing commands in a file called Makefile.

The makefile contains rules which specify what prerequisites are needed to build a particular executable and the steps required to build it. We have multiple Makefiles in this project to recursively compile and build each of our kernel components. In fact, the Makefiles by themselves account for 25% of the codebase.

The Makefiles are executed by issuing corresponding make commands (for eg: make install or make installheaders).Shell scripts are being used to drive the make program. There is a shell script called config.sh that sets up the system environment variables that are utilized by make.

Other shell scripts are being used in the future slides. The compilation process installs the operating system into the system root directory. This is currently the directory sysroot within the project. sysroot directory further contains two sub-directories - boot and usr.

The boot directory contains DaxOS.kernel file and the usr directory contains the standard C library and unit tests. The kernel is booted by pointing the bootloader to the /boot directory.

The built kernel is tested on qemu by calling the qemu.sh shell script. This works by first packaging the kernel into a .iso image file and running qemu with the iso file.

Keyboard design decision

There are two design choices for the implementation of the keyboard driver:

1. Polling Driven

In Polling, the CPU keeps checking the status of the PS/2 keyboard constantly to detect a keypress. Command-ready bit indicate that the device needs servicing. CPU waste lots of CPU cycles by checking the command-ready bit of every device constantly.

2. Interrupt Driven

When the user presses a key, the keyboard driver generates an interrupt which causes the CPU to run and ISR that handles the keypress. Interrupts are signalled by the interrupt-request line. CPU is only called when any device interrupts it.

Interrupt driven implementation is more efficient than polling driven keyboard drivers. Therefore interrupt driven keyboard drive will be implemented .The USB keyboard will be supported by emulation.

Memory segmentation