**Document Objectives**

Create complete basic operating systems terms and concepts information.

**Table of Content Pages**

INTRO ……………………………………………………………………………………... 2

[User VS Kernel](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#user-vs-kernel) ……………………………………………………………………...

[Typical operating system architecture](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#typical-operating-system-architecture) …………………………………………..

[Monolithic kernel](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#monolithic-kernel) ………………………………………………………………….

[Micro kernel](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#micro-kernel) ……………………………………………………………………….

[Micro-kernels vs monolithic kernels](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#micro-kernels-vs-monolithic-kernels) ……………………………………………….

[Address space](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#address-space) ………………………………………………………………………

[User and kernel sharing the virtual address space](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#user-and-kernel-sharing-the-virtual-address-space) ………………………………….

[Execution contexts](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#execution-contexts) ………………………………………………………………….

[Multi-tasking](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#multi-tasking) ………………………………………………………………………

[Preemptive kernel](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#preemptive-kernel) ………………………………………………………………….

[Pageable kernel memory](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#pageable-kernel-memory) ……………………………………………………………

[Kernel stack](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#kernel-stack) ………………………………………………………………………..

[Portability](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#portability) …………………………………………………………………………

[Asymmetric MultiProcessing (ASMP)](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#asymmetric-multiprocessing-asmp) …………………………………………...

[Symmetric MultiProcessing (SMP)](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#symmetric-multiprocessing-smp) ………………………………………………

[CPU Scalability](https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#cpu-scalability) …………………………………………………………………..

Abbreviation ……………………………………………………………………...

**INTRO**

An **operating system** (**OS**) is system software that manages hardware, software resources, and provides common services for computer programs. The application programs make use of the operating system by making requests for services through a defined application program interface (API). In addition, users can interact directly with the operating system through a user interface, such as a command-line interface (CLI) or a graphical UI (GUI).

All types of bare metal may come up with embedded software (hardware specific) or may need installation of generic type of OS in order to function and work properly.

The operating system might be different based on the type of hardware we want to use. The hard ware could be desktop computer, mobile phone, network switch, server, smart TV etc.

But all types of OS do follow similar standards and concepts. Almost all OS provide the following services such as virtual memory management, process scheduling, IPC, network stack etc.

**Why use an operating system?**

An operating system brings powerful benefits to computer software and software development. Without an operating system, every application would need to include its own UI, as well as the comprehensive code needed to handle all low-level functionality of the underlying computer, such as disk storage, network interfaces and so on.

**Operating system types**

*General purpose operating system*: - Examples could be Windows, Linux, Mac, Unix

*Mobile operating system*: - Examples could be Apple iOS and Google Android.

*Embedded operating system*: - A huge assortment of dedicated devices -- including home digital assistants, automated teller machines (ATMs), airplane systems, retail point of sale (POS) terminals and internet of things (IoT) devices -- includes computers that require an operating system. The principal difference is that the associated computing device only does one major thing, so the OS is highly stripped down and dedicated to both performance and resilience.

*Network operating system*: - is another specialized OS intended to facilitate communication between devices operating on a local area network

*Real time operating system*: - When a computing device must interact with the real world within constant and repeatable time constraints, the device manufacturer may opt to use a real-time operating system. E.g FreeRTOS and VxWorks.

**Operating system VS kernel**

**Focus Area**

In this documentation, we will focus mainly on kernels related to general purpose operating system with x86 CPU architecture.

**User VS Kernel**

The kernel is the part of the operating system that runs with higher privileges while user (space) usually means by applications running with low privileges.

These terms are heavily overloaded and might have very specific meanings in some contexts.

User mode and kernel mode are terms that may refer specifically to the processor execution mode. Code that runs in kernel mode can fully control the CPU while code that runs in user mode has certain limitations. For example, local CPU interrupts can only be disabled or enable while running in kernel mode. If such an operation is attempted while running in user mode an exception will be generated and the kernel will take over to handle it.

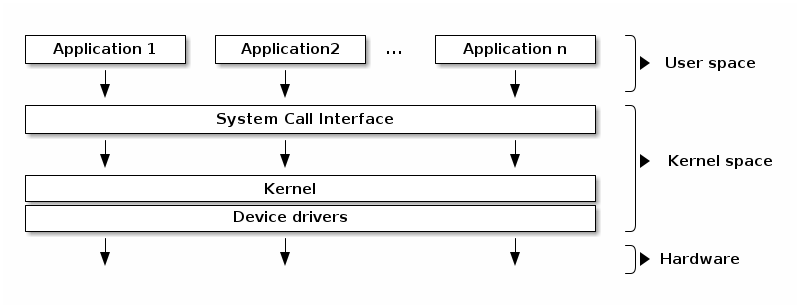
Some processors may have even higher privileges than kernel mode, e.g. a hypervisor mode that is only accessible to code running in a hypervisor (virtual machine monitor)

User space and kernel space may refer specifically to memory protection or to virtual address spaces associated with either the kernel or user applications.

The kernel space is the memory area that is reserved to the kernel while user space is the memory area reserved to a particular user process. The kernel space is accessed protected so that user applications cannot access it directly, while user space can be directly accessed from code running in kernel mode.

**Typical operating system architecture**

In the typical operating system architecture the operating system kernel is responsible for access and sharing the hardware in a secure and fair manner with multiple applications.



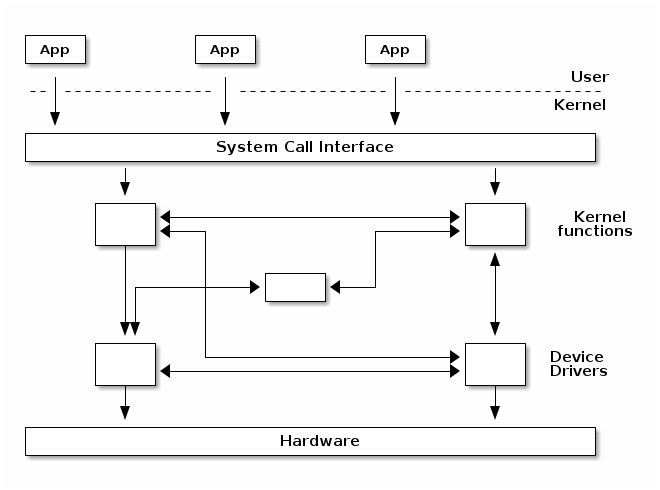
The kernel offers a set of APIs that applications issue which are generally referred to as "System Calls". These APIs are different from regular library APIs because they are the boundary at which the execution mode switch from user mode to kernel mode.

The kernel code itself can be logically separated in core kernel code and device drivers code. Device drivers code is responsible of accessing particular devices while the core kernel code is generic. The core kernel can be further divided into multiple logical subsystems (e.g. file access, networking, process management, etc.)

### Monolithic kernel

A monolithic kernel includes all (or at least, most) of its services in the kernel. This reduces the amount of context switches and messaging involved, making the concept faster than a [Microkernel](https://wiki.osdev.org/Microkernel). On the downside, the amount of code running in kernel space makes the kernel more prone to fatal bugs.

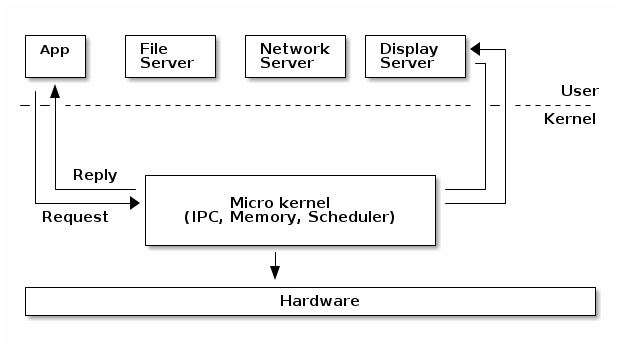
A monolithic kernel is one where there is no access protection between the various kernel subsystems and where public functions can be directly called between various subsystems.



However, most monolithic kernels do enforce a logical separation between subsystems especially between the core kernel and device drivers with relatively strict APIs that must be used to access services offered by one subsystem or device drivers.

### Micro kernel

A micro-kernel is one where large parts of the kernel are protected from each-other, usually running as services in user space. Because significant parts of the kernel are now running in user mode, the remaining code that runs in kernel mode is significantly smaller, hence micro-kernel term.



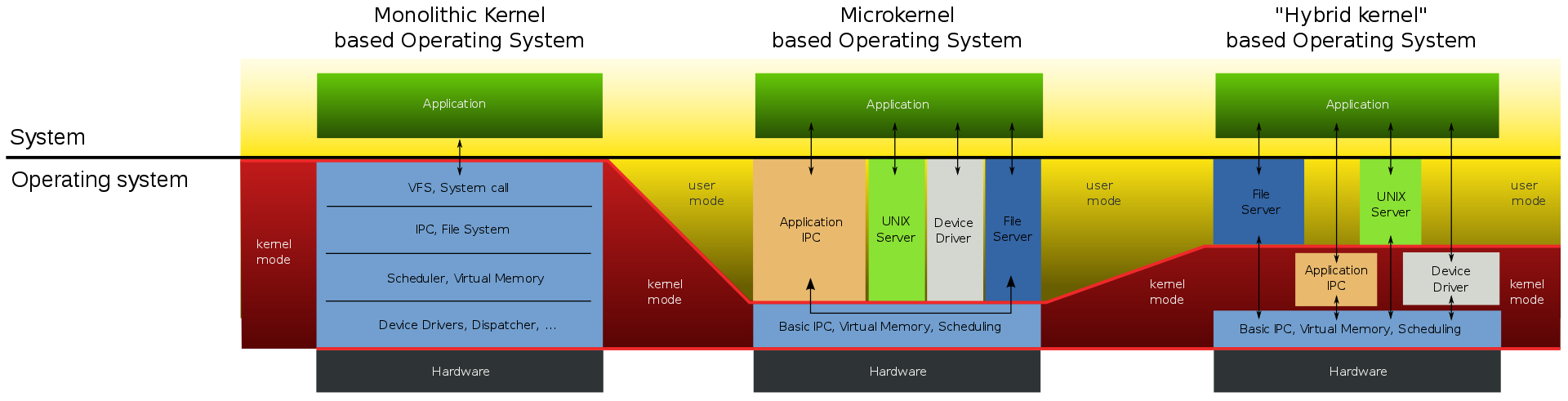
In a micro-kernel architecture the kernel contains just enough code that allows for message passing between different running processes. Practically that means implement the scheduler and an IPC mechanism in the kernel, as well as basic memory management to setup the protection between applications and services.

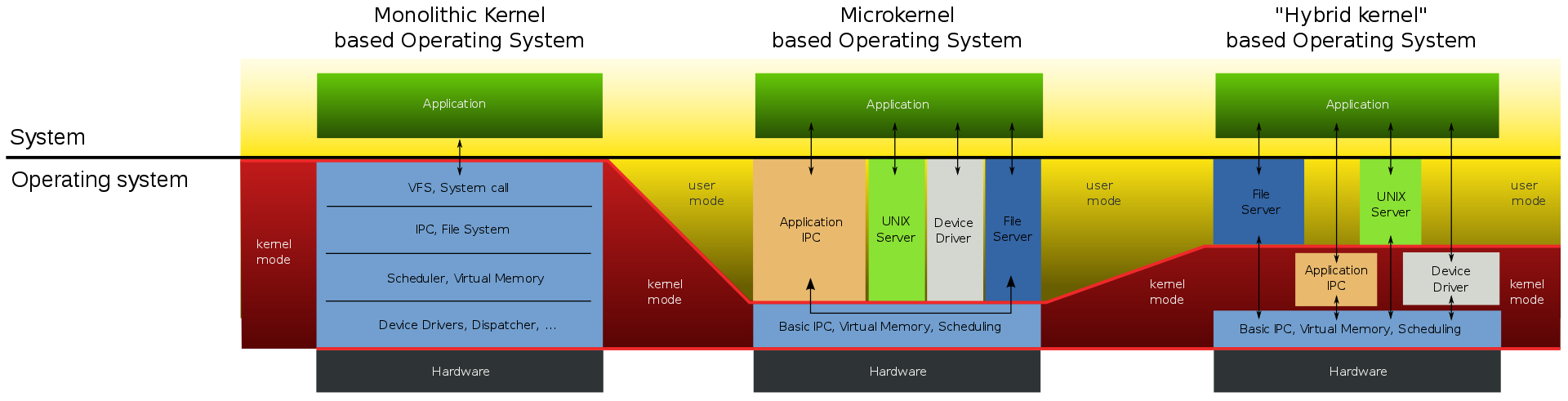
One of the advantages of this architecture is that the services are isolated and hence bugs in one service won't impact other services.

As such, if a service crashes we can just restart it without affecting the whole system. However, in practice this is difficult to achieve since restarting a service may affect all applications that depend on that service (e.g. if the file server crashes all applications with opened file descriptors would encounter errors when accessing them).

This architecture imposes a modular approach to the kernel and offers memory protection between services but at a cost of performance. What is a simple function call between two services on monolithic kernels now requires going through IPC and scheduling which will incur a performance penalty

### Micro-kernels VS Monolithic kernels





**Abbreviation**

IPC: - Inter process Communication

**Reference**

<https://wiki.osdev.org/>

<https://linux-kernel-labs.github.io/refs/heads/master/so2/lec1-intro.html#user-vs-kernel>

**Bad references**

[Difference Between Microkernel and Monolithic Kernel (with Comparison Chart) - Tech Differences](https://techdifferences.com/difference-between-microkernel-and-monolithic-kernel.html)