Freie Universitat Berlin
Erasmus Program

10 ECTS

Systems Software

Professor:
Prof. Barry Linnert

Autor:
Filippo Ghirardini

Winter Semester 2024-2025

Contents

Systems Software

Autor: Ghirardini Filippo

Winter Semester 2024-2025

---

1 Introduction

Definition 1.0.1 (Operating system)**.** *The programs of a digital computing system which lay, together with the basic properties of the computing system, the foundation for the possible modes of operation and especially control and monitor the execution of programs.*

The main tasks of an OS are:

- Provision of virtual machine as an abstraction of the computer system
- Resource management
- Adaptation of machine structure to user requirements
- Foundations for a controlled concurrency activities
- Management of data and programs
- Efficient usage of resources
- Support in case of faults and failure

All of these tasks must be taken care of by the OS architect while new hardware and new applications come out in a complex market.
Every complex system in every area (e.g. buildings) is composed of single components of different types. Successful design of a complex system requires the knowledge of different variants of the components and their interplay.

1.1 History

1.1.1 1950

In the fifties only one program was executed by one processor. The OS functionality is limited to support for input/output and transformation of number and character representation.

1.1.2 1960

The CPU and I/O speed become **faster**. Now the OS supports multiple applications running at the same time with real **parallelism**. The notion of process as a virtual processor is born and the memory is now virtualized. Interactive operation by more than one user (**timesharing**).

1.1.3 1970

Software crisis: OS become large, complex and error prone. Now design, maintainability and security are important (software engineering). High level programming languages are now used to program OS. The need for modular programming, abstract data types and object orientation come up.

1.1.4 1980

Personal computers are born. The systems can now be **connected**, for example via Ethernet. Processes are now fundamental and complex: since a context switch is very CPU intensive, address space and processes are now separated to allow sharing of the address space (**threads**).
Parallelism becomes a part of programming languages. There is a need to create standards and protocols.

1.1.5 1990

Due to the popularity, microprocessors become **cheap**. Now multiple chips can work together. GUIs are born and with them **audio and video data**. Birth of the **Web**.

1.1.6 2000

Now everything is a computer: it's ubiquitous and pervasive. The cloud computing is born. Today the main topics are:

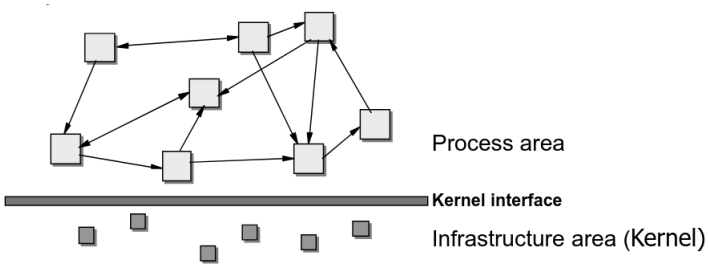
- Safety and security
- Robustness and dependability
- Virtualization
- Optimization for multi core processors (scheduling, locking)
- Energy consumption
- User interface
- Database support for file systems
- Cluster-Computing, Grid-Computing and Cloud-Computing
- Small OS for small devices

2 Architecture

A general system consists of **elements** and **relationships** between them. The elements are the functional units with interaction of different kinds in between (e.g. data flow, request flow, call, etc.).

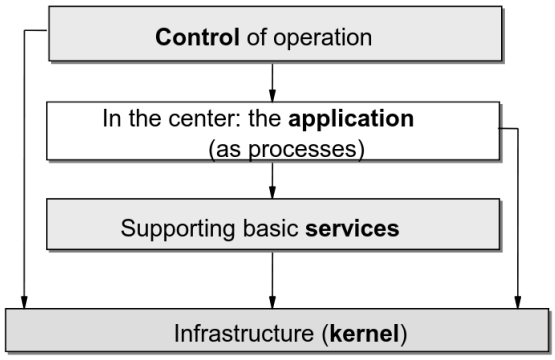
In an **operating system** the elements are the **processes** and they interact with each other.. Since they are not hardware native we need something that provides them and also the interactions: the **kernel.**. Therefore, we'll have two areas:

- *Process area*: where the OS functionalities are located
- *Kernel area*: provides the fundamental infrastructure for the processes



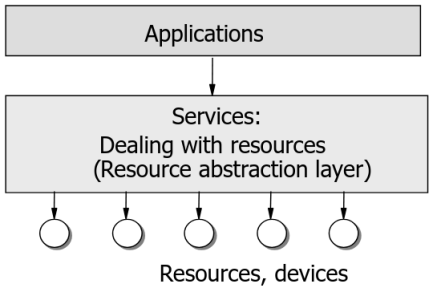
2.1 Process area

In more details, the process area is divided like this:



2.1.1 Services

In particular, services deal with **resources** needed by applications, since handling them directly is very tedious.

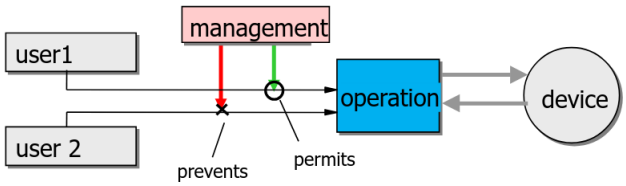


Therefore, we need to make a distinction between resources:

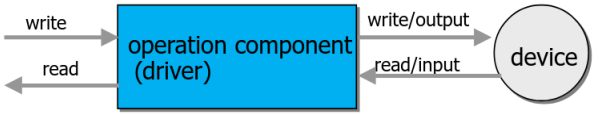
- **Logical**: resources for made organizational reasons by real ones (e.g. files)
- **Physical**: real existent (e.g. mouse, keyboard)

The main two aspects of dealing with resources are:

- **Management**: in case of competition, deciding who should get the resource and when



- **Operation**: real usage (e.g. data transport) with **read** and **write** operations



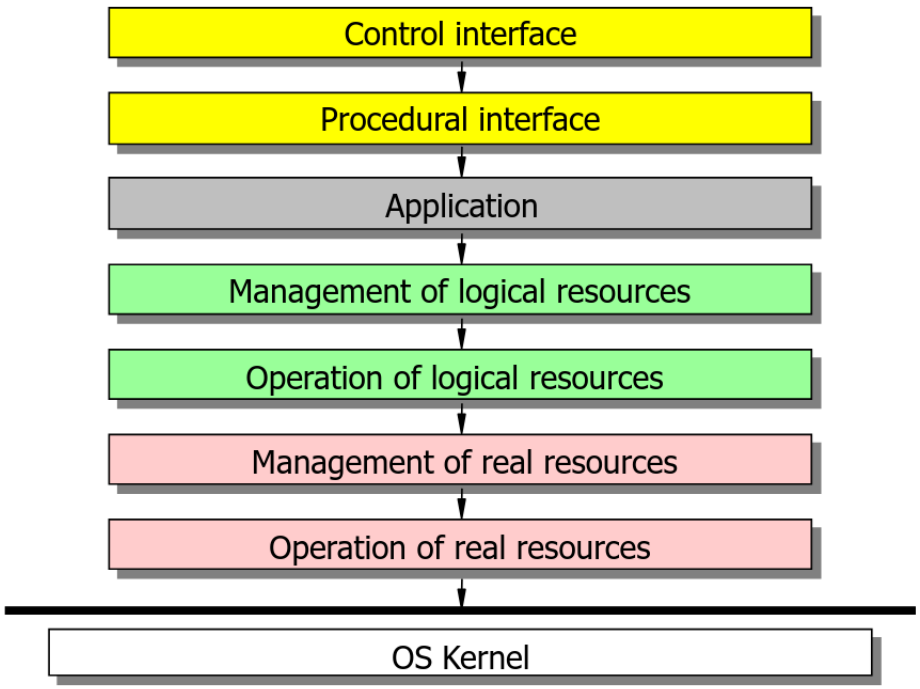
In the end we have both the *management* and the *operation* for the *logical* resources and for the *real* ones.

2.1.2 Control

We distinguish two interfaces:

- **Control** interface: handles the interactions between the user and the machine (OS commands and UI)
- **Procedural** interface: has the possibility to make complex requests to the OS, also with programming language notation

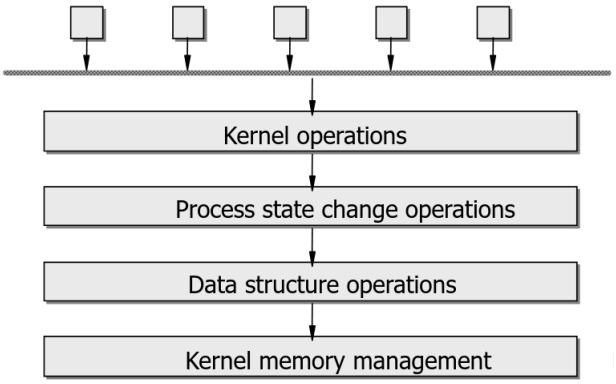
2.1.3 Overview



Note 2.1.3.1. Each layer may be **partitioned** and upward calls are allowed as long as there are no cycles.

2.2 Kernel area

In details, the kernel area is divided like this:



There are two ways of realizing a kernel:

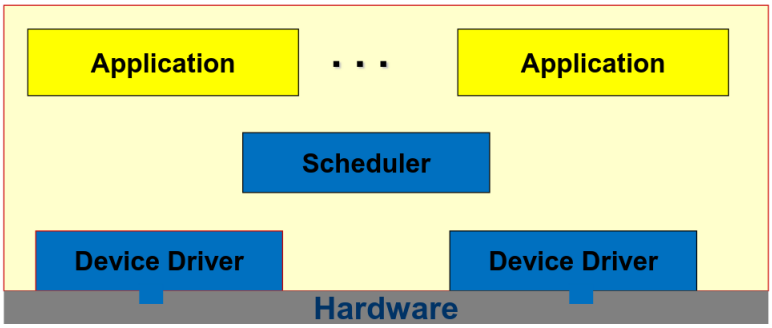
- **Scattered** across programs: kernel operations resides in different program address spaces
- **Compact**: there is the kernel address space which contains its own procedures

2.2.1 Size

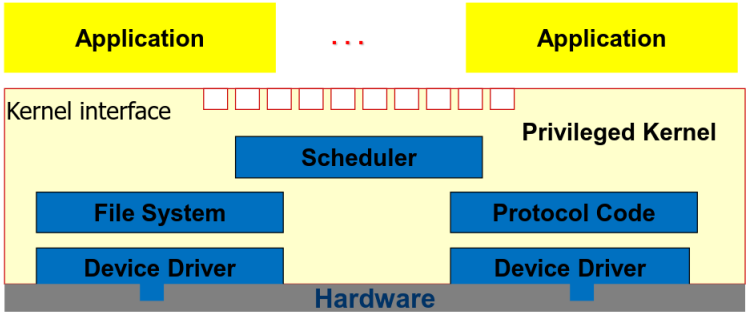
Depending on what functionalities you add in the kernel, you may end up with different kernel sizes. The bare minimum is the one described above (process management and communication) and it's called **microkernel**.

2.2.2 Monolithical

Another approach is the **monolithical system**, where there is no strict separation between applications and OS. It's appropriate for small OS.

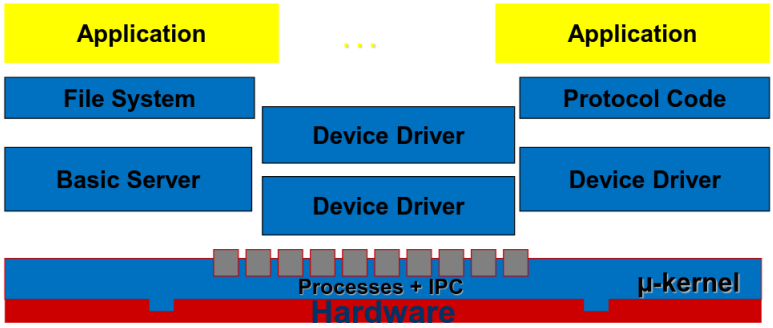


On the other hand, **monolithical OS-kernels** have a separation between applications and OS for **protection** reasons but no separations among OS components.



2.2.3 Microkernel

As stated before, a microkernel contains only process management (initializing and dispatching) and Inter Process Communication.



The **advantages** are:

- Supports **modular** structure
- Since the services are outside of the kernel, there is more **stability** and **security** because it will not be affected by faulty services. Furthermore it improves **flexibility** and **extendibility** since services can be removed and added arbitrarily.
- The safety-critical part (kernel) is **small** and easily verifiable
- Usually only the kernel needs to run in **privileged** mode
- It allows the coexistence of several **interfaces**

The main **disadvantage** is that there are **performance** issues since interplay of components outside the kernel need more IPC and therefore more kernel calls.

2.3 Design principles

The basic idea is KISS: Keep It Simple and Stupid.

2.3.1 Modularity

The system is decomposed in a set of modules so that:

- the **interaction** (information and control flow) within the module is high
- the **interaction** between modules is low
- the **interfaces** between them are simple
- the modules are small and easily understandable

2.3.2 Hierarchization

Homogeneous elements are grouped together in a tree structure to guarantee **scalability** and mastering complexity.

2.3.3 Layering

System functionalities should be divided into layers, with the simpler one at the bottom. Each new layer represents an abstraction of the previous ones and provides an interface for the upper layers.

Esempio 2.3.1. An example of layering is the Internet protocol stack.

2.3.4 End to end

A function of service should be carried out within a general layer only if it is **needed** by all clients of that layer and if it can be completely **implemented** in that layer.
In the OS context this means a **stable**, **universal** programming interface should be provided. Support should be placed in the upper layer.

Application neutrality The OS should be application neutral, meaning that lower layers may provide mechanisms that can be parameterized in higher layers to suit specific application requirements (e.g. scheduling, paging).

Orthogonality Function and concepts should be independent. Each component should be orthogonal: freedom of combination.

SPOT Single Point of Truth is a rule to avoid inconsistencies by avoiding repetitions in code (only one implementation) and in data (only one representation).