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:

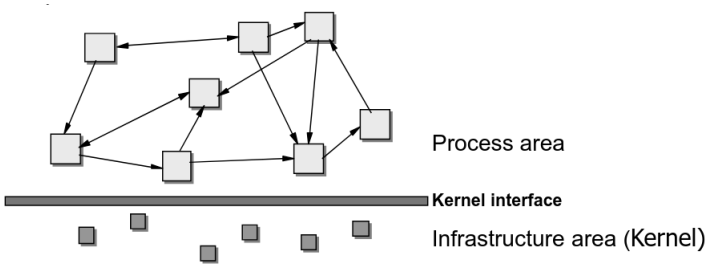
- Safety and security
- Robustness and dependability
- Virtualization
- Optimization for multi core processors (scheduling, locking)
- Energy consumption
- User interface
- Database support for file systems
- Cluster-Computing, Grid-Computing and Cloud-Computing
- Small OS for small devices

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.).

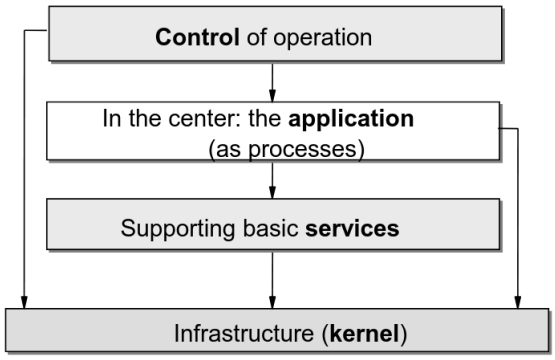
In an **operating system** the elements are the **processes** and they interact with each other.. Since they are not hardware native we need something that provides them and also the interactions: the **kernel.**. Therefore, we'll have two areas:

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



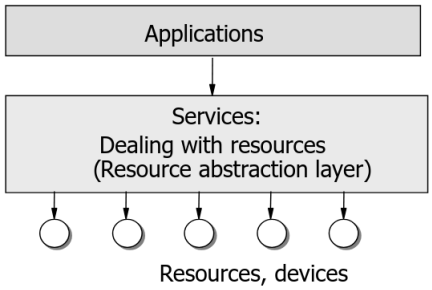
2.1 Process area

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.

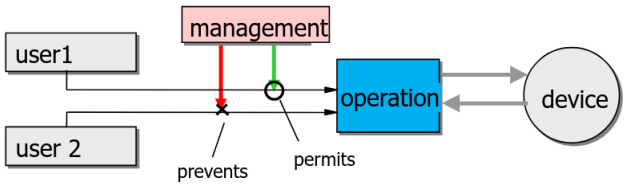


Therefore, we need to make a distinction between resources:

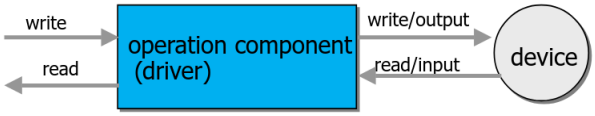
- **Logical**: resources for made organizational reasons by real ones (e.g. files)
- **Physical**: real existent (e.g. mouse, keyboard)

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



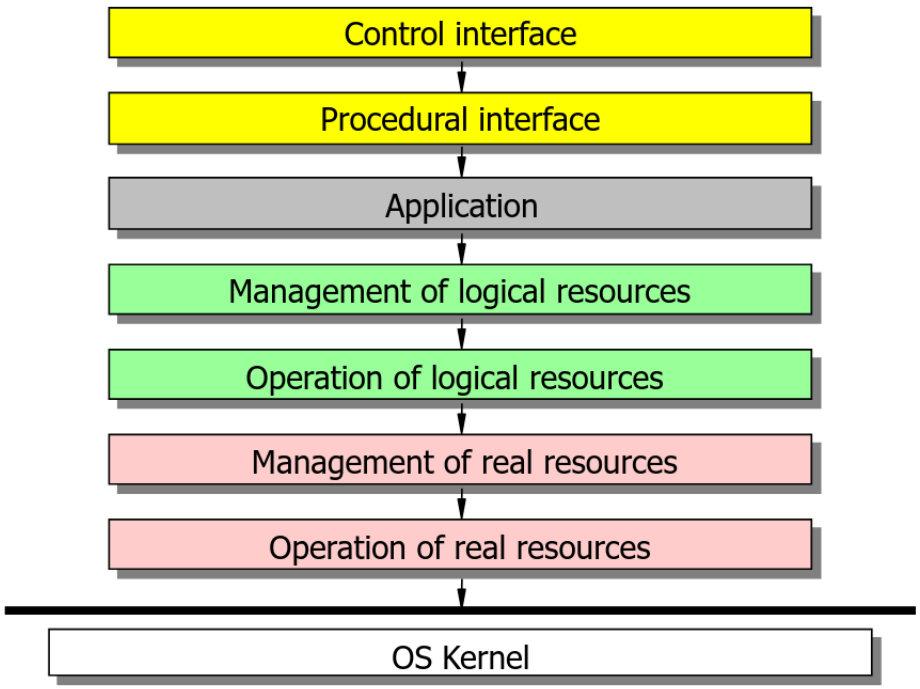
In the end we have both the *management* and the *operation* for the *logical* resources and for the *real* ones.

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

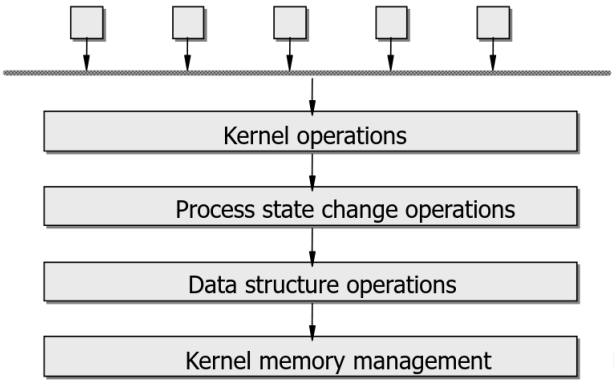
2.1.3 Overview



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:

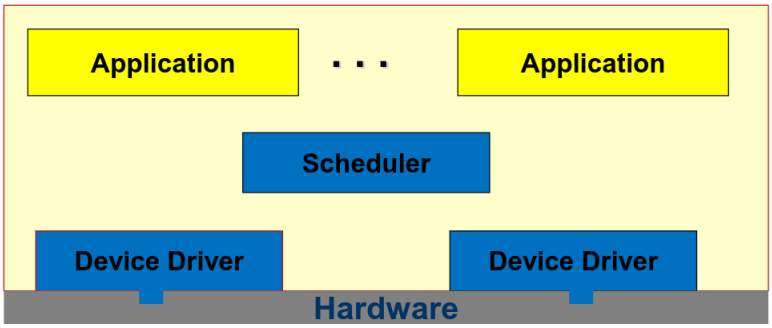
- **Scattered** across programs: kernel operations resides in different program address spaces
- **Compact**: there is the kernel address space which contains its own procedures

2.2.1 Size

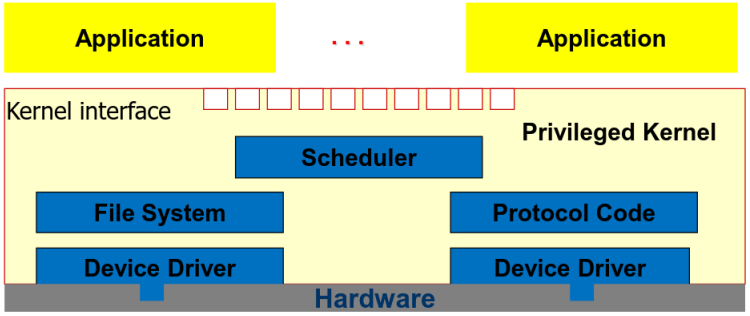
Depending on what functionalities you add in the kernel, you may end up with different kernel sizes. The bare minimum is the one described above (process management and communication) and it's called **microkernel**.

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.

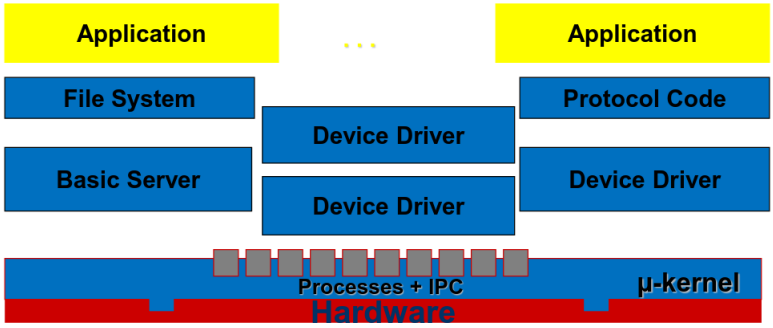


On the other hand, **monolithical OS-kernels** have a separation between applications and OS for **protection** reasons but no separations among OS components.



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).

3 Threads

3.1 Description

3.1.1 Thread Control Block

A thread is represented in the memory by the **Thread Control Block**, a structure that contains all the relevant information like:

- Thread **characteristics**: thread ID, name of the program
- **State information**: instruction counter, stack pointer, register contents
- **Management** data: priority, rights, statistics

3.1.2 Management

Big systems can have thousands of threads, therefore we need efficient ways to manage them. There are different options:

1. Single threads not connected to each other
2. All the threads are in a static long array
3. All the threads are in a linked list of variable length
4. Tree
5. The threads are managed via an inverted table

In general, to increase **efficiency**, threads that forms a subset regarding a particular attribute (e.g. thread state) should be grouped together.

3.1.3 Static and dynamic OS

Depending on the OS type, threads may be managed differently:

- **Static OS**: all threads (one per application) are statically defined in advance and the TCBs are declared as variables.
- **Dynamic OS**: the threads are created and deleted by kernel operations

3.1.4 Address Space

A logical address space of a thread is the set of its valid addresses that it can access. Modern CPUs allow **relative addressing** and address translation via a Memory Management Unit. This allows to have an arbitrary number of logical addresses mapped to a physical address space.
This also leads to mutual protection since the address spaces are **independent**, while at the same time allowing **relations**:

- A thread owns one **private address** space (Unix process)
- Several threads **share** an address space (Thread)
- A thread **switches** from an address space to another

3.2 Switch

Thread switching means that the processor stops the execution of the current thread and continues with the execution of another. It's the transition from one instruction sequence to another.

3.2.1 Jumping

Switching by jumping means using a *jump* instruction (programmed statically) to jump to another thread. A switching point consists of:

- **Continuation address**: where we interrupted the work
- **Jump instruction**

The only use case for this type of approach is for very specific real time applications, to reduce as much as possible the latency. At the same time it's very **inflexible** since we often don't know from where we return , which one is the next thread and we often need the values of the registers.

3.2.2 Context switch

Continuation address The first phase is to store the address of the next instruction to be executed in a special variable **ni**, located in the TCB.

Selection of next thread The next thread to be executed is often determined at the moment of the switching, following different criteria:

- **Cycling switching**: number of thread
- Order of arrival
- **Priority**: in this case the order can be organized in two dimensions, where each row is a group of threads with the same priority. It can be:
  - Constant
  - Dynamic

The selection problem can be solved inserting the threads in the selected order at the arrival.

Context switching The *jump* switch loses the content of the registers. In this case, we save the **state of execution** (arithmetic registers, index registers, processor state, etc..) and the **thread execution environment** (address registers, segment tables, interrupt masks, etc..) as a **thread context**.
This is the most consuming part of the switch, therefore there are special instructions and registers to speed up the process

Osservazione 3.2.1 (Subroutine)**.** To avoid redundant calls, we can use the switch as a common **subroutine**. In this way, when a thread calls for the switch, the second thread can come back directly to the same subroutine avoiding the save of the *continuation address*.

Osservazione 3.2.2 (Thread control)**.** Assuming the paradigm of **cooperative scheduling**, when the first thread calls for the switch subroutine, both the caller and the next thread need to give up and get the control at the same time. This is done by changing **stack pointer**.

3.3 Conditioned switching

When a thread needs to wait for something to happen, the processor could do some other work instead of waiting. Therefore this happens when a certain condition is met.

3.3.1 Thread states

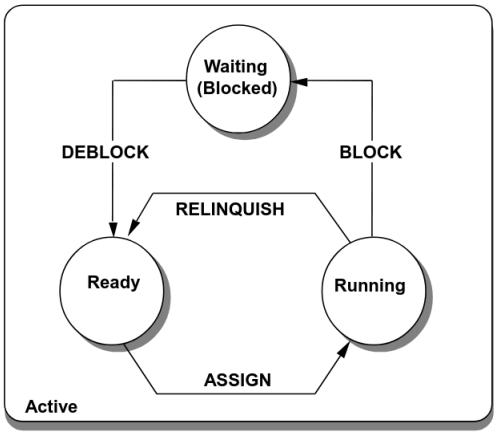
When a thread is waiting and we choose to switch to another one, we may encounter a thread that's also waiting and so on. To speed up the search for a working thread we divide them by **states**:

- **Running**: threads that are currently being executed
- **Ready**: threads that are ready to be executed and are waiting for the processor

- **Waiting**: threads that are blocked and waiting for some external event

We handle state changes via these transitions:

- **Relinquish**: voluntarily switching to another thread
- **Assign**: taking the next thread from the ready set to resume its operation on the processor
- **Block**: leave the processor since the condition is not met and must not be resumed until then
- **Deblock**: if the event happened for which the blocked thread waited it changes its state from waiting to ready and is inserted into the set of ready threads



Note 3.3.1.1. State change operations may also include other actions such as a thread switch. E.g. for the *relinquish*.

3.4 Automatic switching

In many cases it's not viable to insert a switch call inside the thread. In this case we need an automatic way of doing that through a **clock** that specifies:

- The deadline for each thread, called a **time slice**
- Interrupt on timeout

Since in this case it's the kernel that handles the switch, it needs to save the stack pointer, the interrupt pointer and the flags of the current thread to its own stack. Doing so, it can be pushed back again when it's his turn.

3.4.1 Kernel exclusion

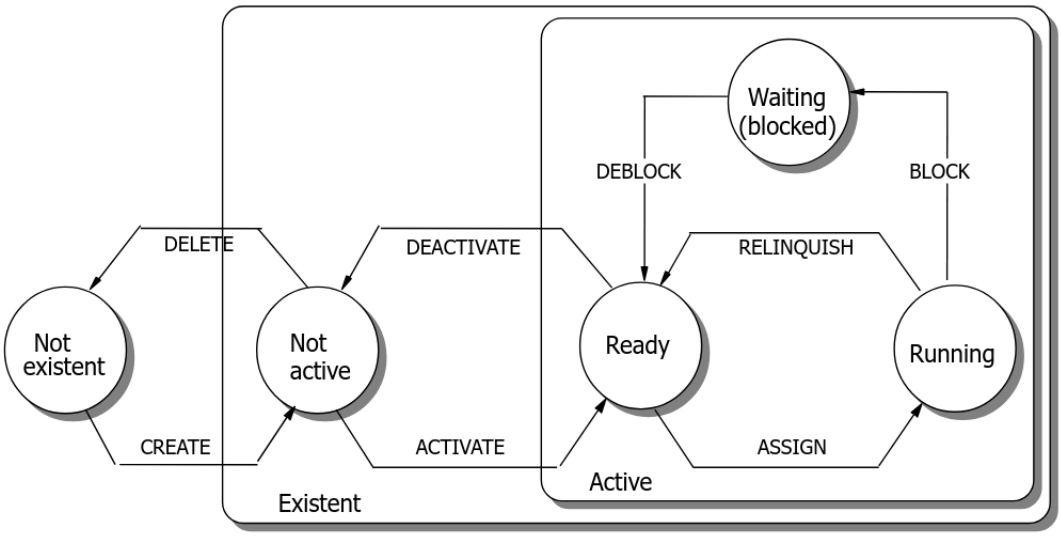
When working with automatic switching we need to make sure that during the switch no other kernel operations are done, since they all work on the same shared data structures. This could cause faulty behaviors, e.g. during a condition evaluation a switch occurs.
To avoid this we use **mutual exclusion**, and in the case of the kernel we handle it entirely as a **critical section**. We distinguish four cases:

- Single processor with no interrupts: we don't have problems
- Single processor with interrupts: we disable interrupts during kernel operations
- Multi processor with no interrupts: we use a **spin lock**, called also **busy waiting**, where we constantly check if the kernel is busy
- Multi processor with interrupts: we disable the interrupts and then we check if the kernel is lock, if that's true we enable them and stay in busy waiting. Otherwise we do the kernel operation and then enable them.

3.5 Dynamic system

In these types of systems the number of threads is variable. Therefore we need the following operations:

- **Activate/Deactivate**: a thread exists (there is a TCB) but it's "resting". We use these operations to change from the state *active* and *inactive*
- **Create/Delete**: if the thread does not exist we need to create and then delete them



3.5.1 Preemption

There may be thread whose execution is really urgent and that cannot wait for a switch to happen. When these threads show up, the switch happens immediately and the first one is being **preempted** by the one with the higher priority.
In practice, we check for preemption whenever a new thread is added to the queue.

3.5.2 Idle

In operating systems with waiting states it may happen that all threads all waiting for something. To handle this we add an **idle thread** that:

- **Never stops**
- has the **lowest priority**
- must be **preemptable** at any time

In practice we could use an empty loop (but wastes energy), a special operation like *halt* that does not access memory but reacts to external signals, or a thread that does useful operations like checks and reorganizations.

3.5.3 Initialization

To switch to a thread for the first time we use the *switch* procedure. To do so we need to prepare the TCB and the Stack as if it was already in the middle of a switch.

3.6 Programming languages

Many modern programming languages contain a thread concept to formulate concurrent activities within programs (e.g. Java Threads), or there are programming libraries that extend it by a thread concept. In these cases the threads are invisible to the OS, which sees the whole program as a thread.