The difference between a good and a poor architect is that the poor architect succumbs to every temptation and the good one resists it.
-- *Ludwig Wittgenstein*
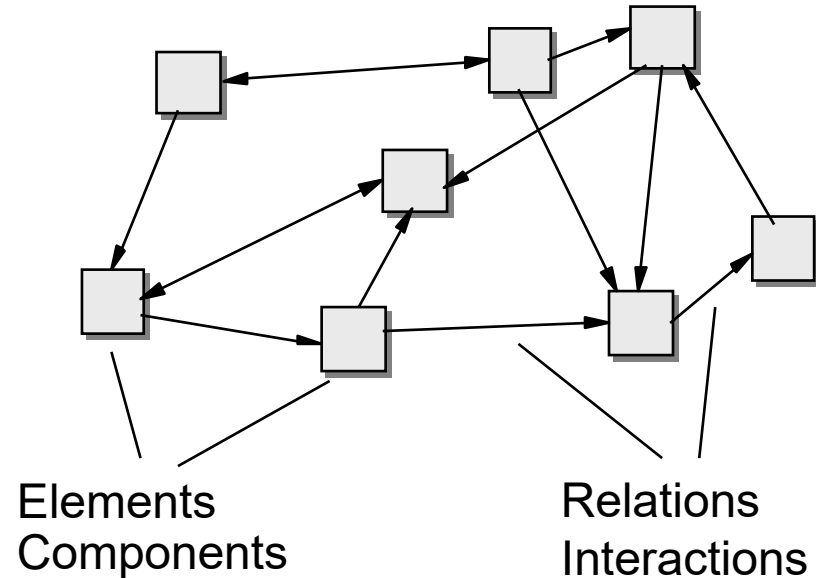
# Chapter 2

Architecture

A system (e.g. operating system, software system) consists of
- Elements
- Relations between elements

Elements
Components

Relations
Interactions

The elements are functional units with interactions of different kinds in between (data flow, request flow, synchronization, call, communication, ...).
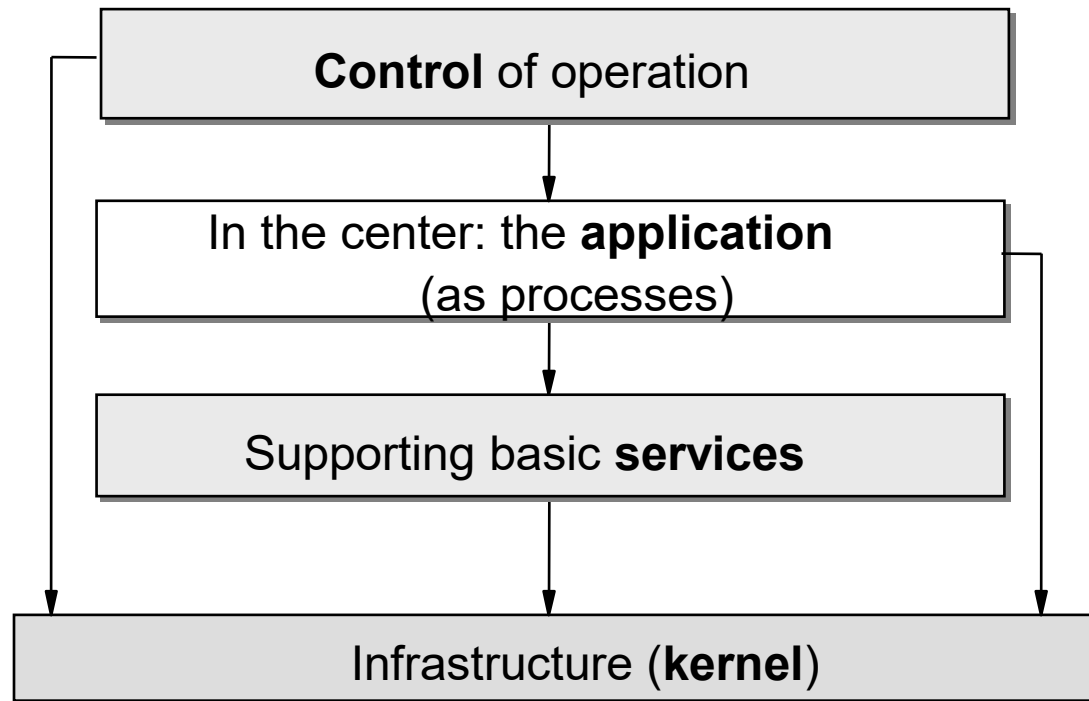
# Coarse division of an OS

- In operating systems the elements/components are the *processes*.
- An operating system consists of a set of interacting processes.
- Since processes are not offered by the hardware, there must be something that provides the concept of a process and the interaction between processes.
- This „something" is called kernel of the operating system. It provides the basic infrastructure for the OS.

Process area

**Kernel interface**
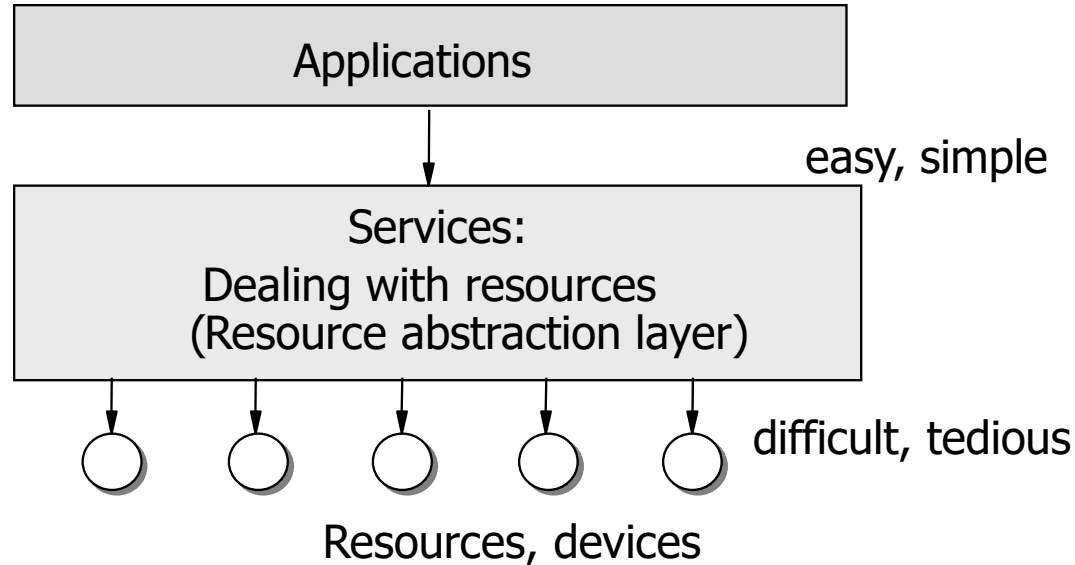
Infrastructure area (Kernel)
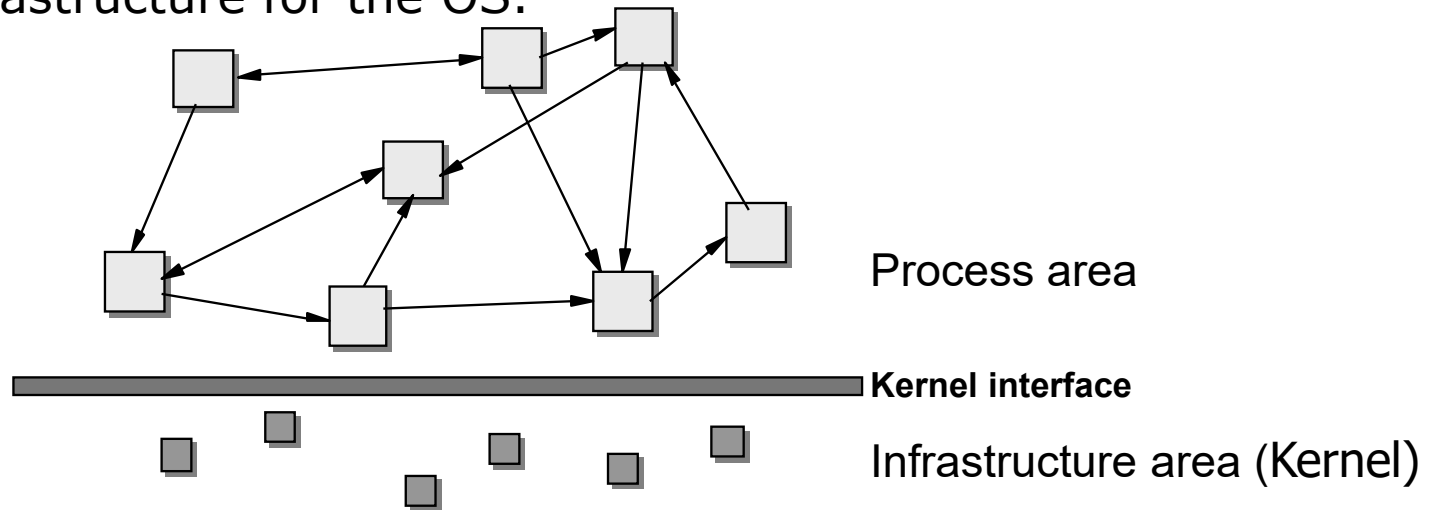
- In a first coarse structure, we therefore distinguish two areas:
  - Process area, where the essential OS functionality is located.
  - Kernel area that provides the fundamental infrastructure for these processes.

Finer resolution: Process area

```
┌─────────────────────────────────────────────┐
│              Control of operation             │
└─────────────────────────────────────────────┘
                        │
                        ▼
┌─────────────────────────────────────────────┐
│          In the center: the application       │
│                (as processes)                 │
└─────────────────────────────────────────────┘
                        │
                        ▼
┌─────────────────────────────────────────────┐
│           Supporting basic services           │
└─────────────────────────────────────────────┘
                        │
                        ▼
┌─────────────────────────────────────────────┐
│           Infrastructure (kernel)             │
└─────────────────────────────────────────────┘
```

# Services

Finer resolution: Services

```
┌─────────────────────────────────────────┐
│              Applications                 │
└─────────────────────────────────────────┘
                    │
                    ▼                    easy, simple
┌─────────────────────────────────────────┐
│              Services:                    │
│          Dealing with resources           │
│      (Resource abstraction layer)         │
└─────────────────────────────────────────┘
     │      │      │      │      │
     ▼      ▼      ▼      ▼      ▼         difficult, tedious
    ◯      ◯      ◯      ◯      ◯
```

Resources, devices

Distinction

| | |
|---|---|
| *Logical* resource: | „thought up" for organizational reasons realized by real resources Examples: file, window |
| *Real* (physical) resource: | real existence, to be touched Examples: disc, keyboard, … |

# How to deal with resources
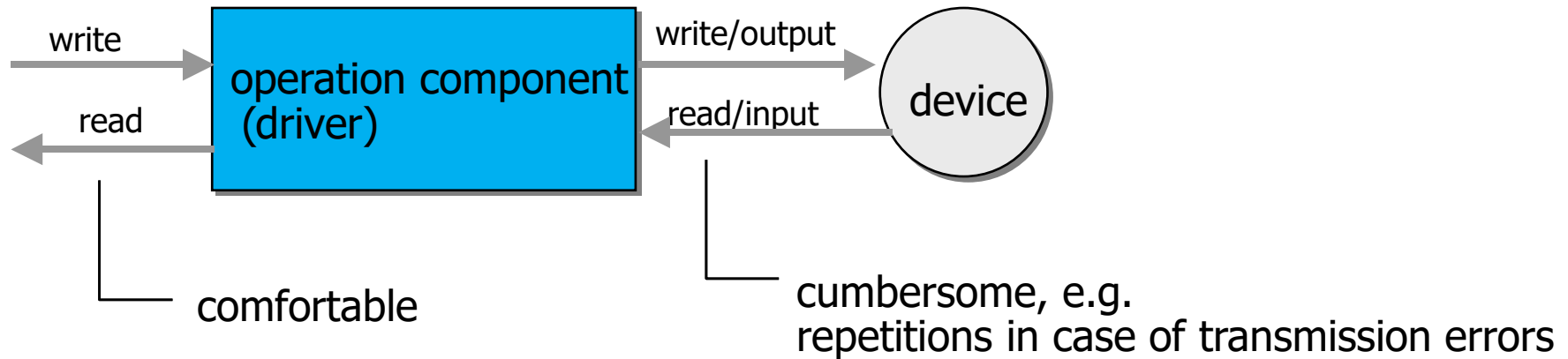
Two aspects:

Resource management:   only in case of competition for resources: who may accesses what and when?

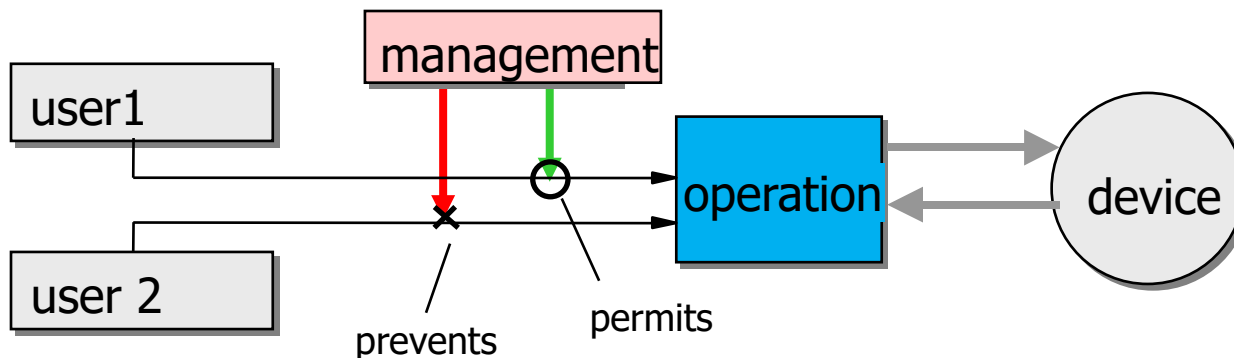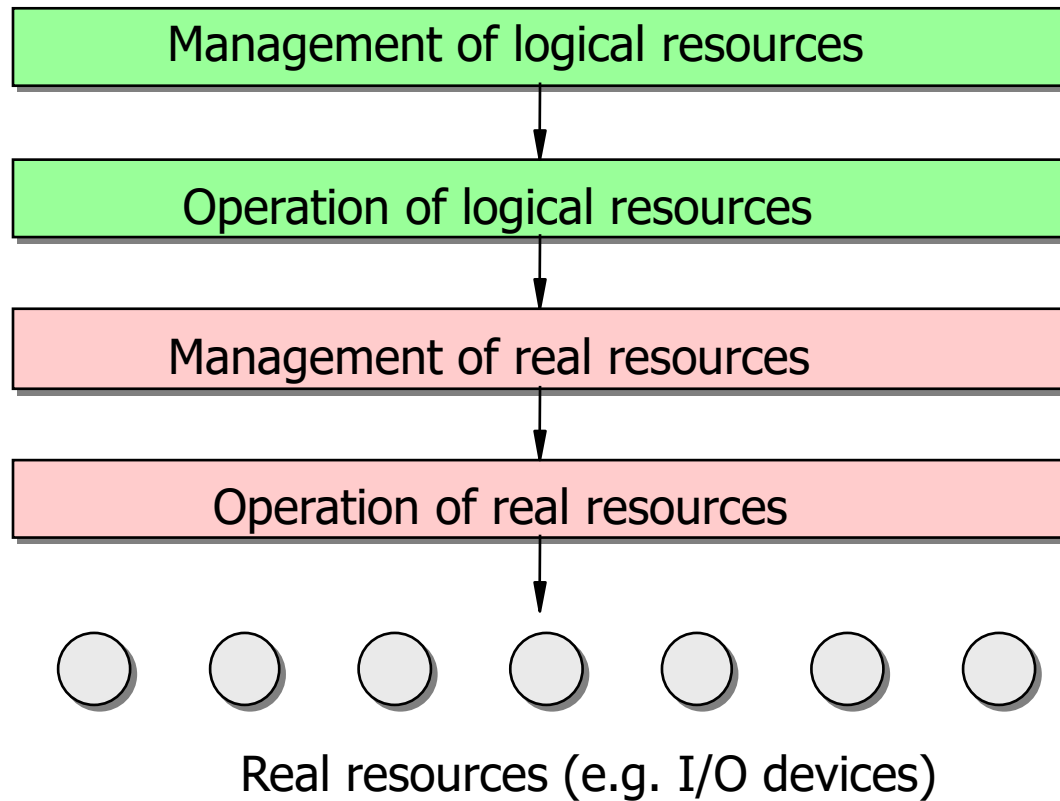Resource operation:      real usage, e.g. data transport
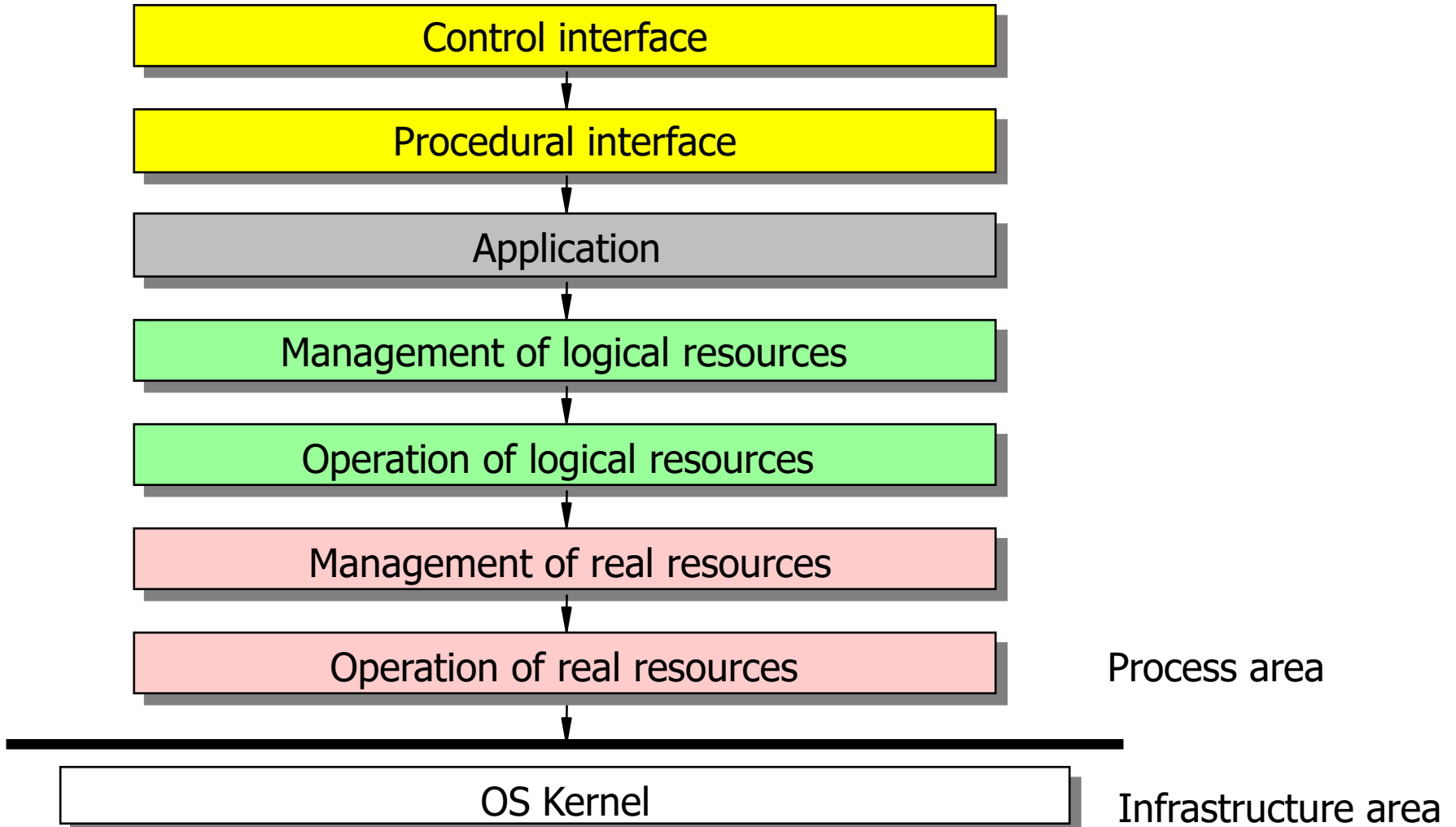
Example:

| Allocate resource | *management* | Rent a car |
|---|---|---|
| Use resource | *operation* | Drive the car |
| Release resource | *management* | Return the car |

# The term "operation"

write → **operation component (driver)** → write/output → device

read ← **operation component (driver)** ← read/input ← device

comfortable

cumbersome, e.g.
repetitions in case of transmission errors

# The term "management"

**management**

user1

user 2

operation → device

prevents

permits

# Operating system services
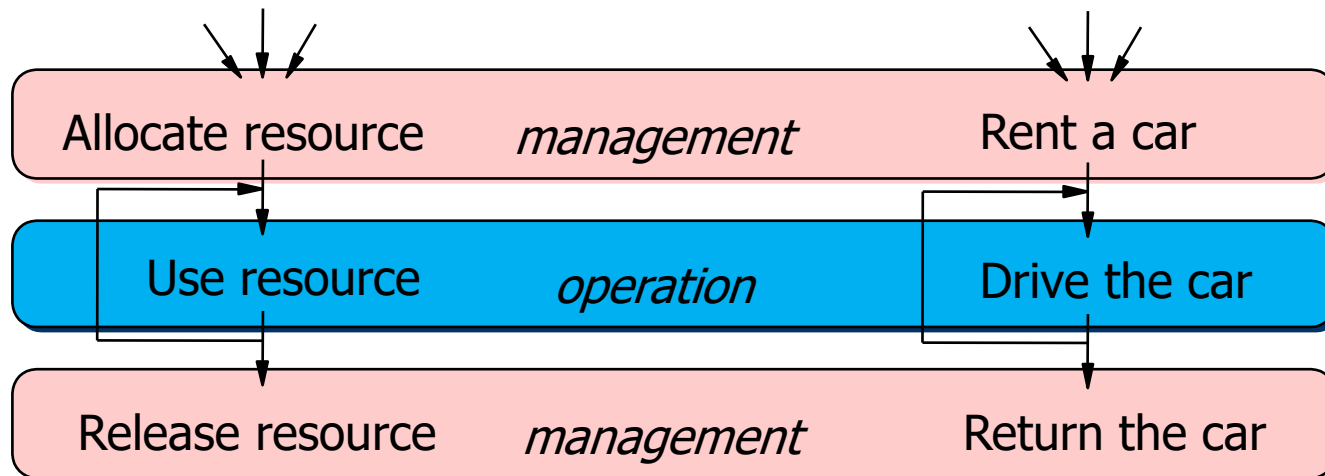
- **Structure of service layer**

| Management of logical resources |
|:---:|

↓

| Operation of logical resources |
|:---:|

↓

| Management of real resources |
|:---:|

↓

| Operation of real resources |
|:---:|

↓

○ ○ ○ ○ ○ ○ ○

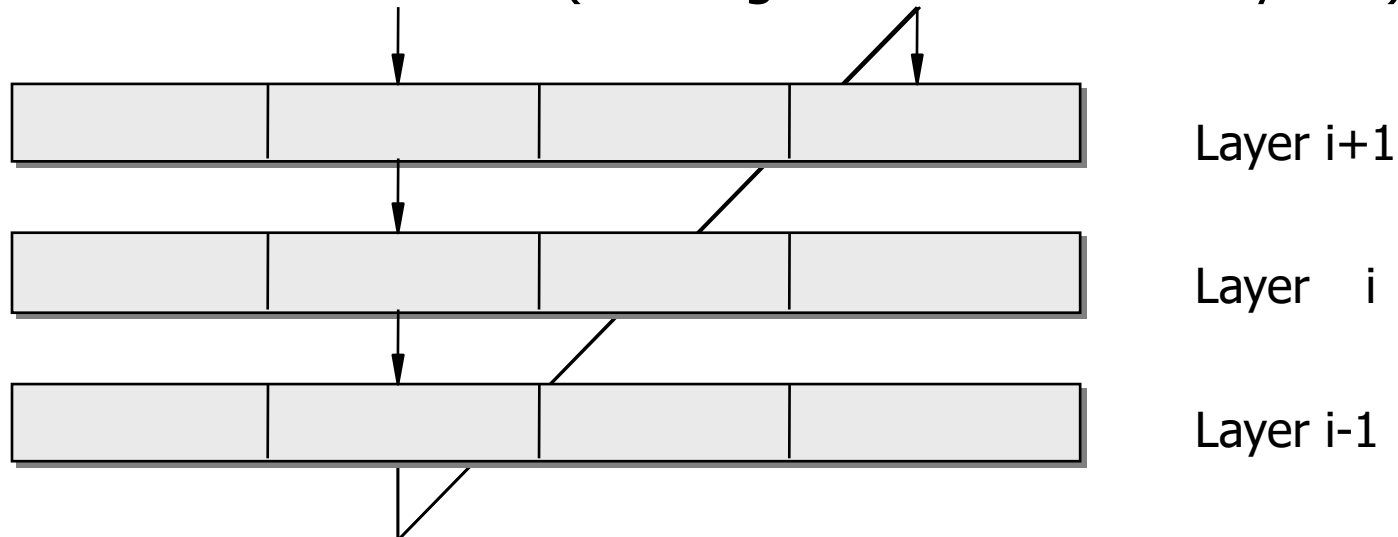Real resources (e.g. I/O devices)

# Control

We distinguish

- Control interface
  - Interaction between human and machine

    - OS-commands

    - User interface (textual, graphical, touch, acoustic, …)

- Procedural interface
  - Possibility to define complex requests to the OS

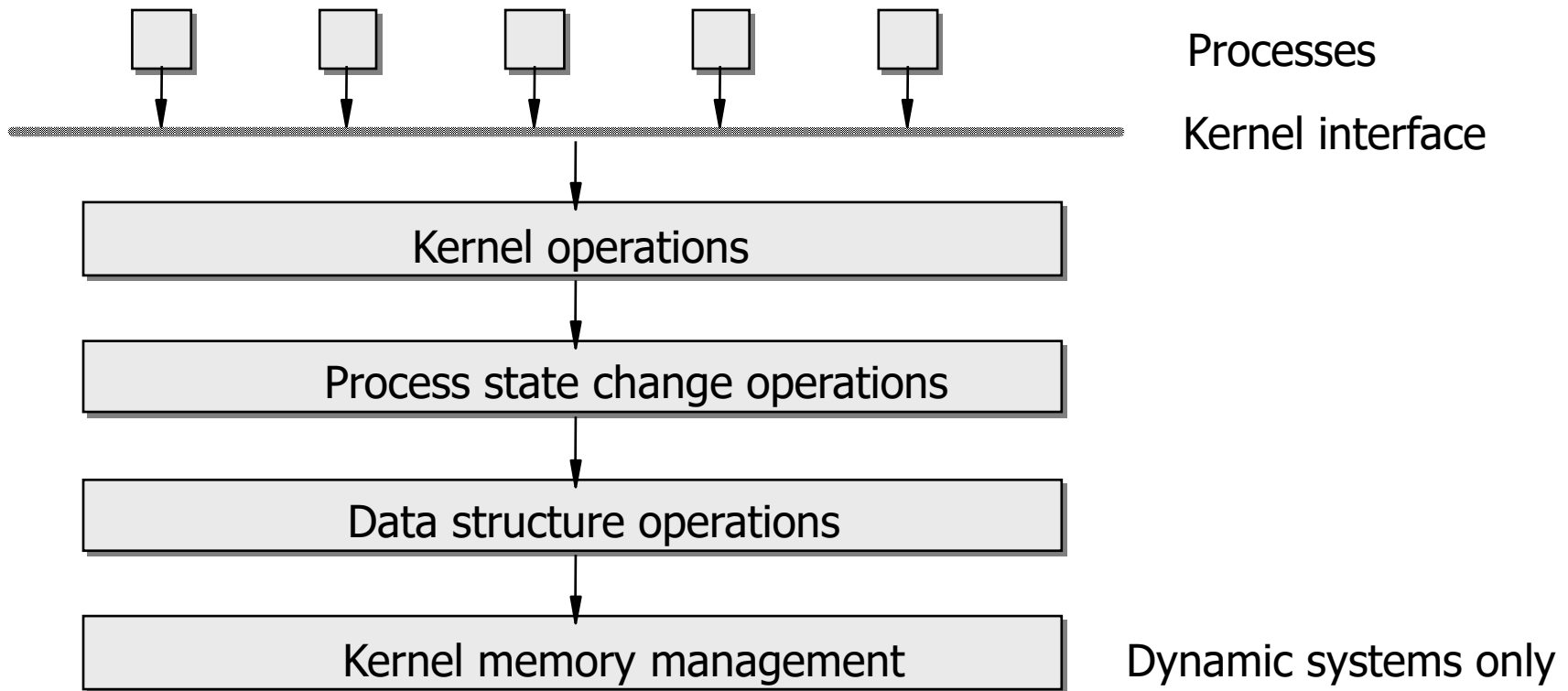  - Programming language notation with embedded OS commands to define and control complex tasks

Control interface

Procedural interface

Application

Management of logical resources

Operation of logical resources

Management of real resources

Operation of real resources — Process area

OS Kernel — Infrastructure area

Each layer may be partitioned.

| Operation device A | Operation device B | Operation device C | Operation device D |
|---|---|---|---|
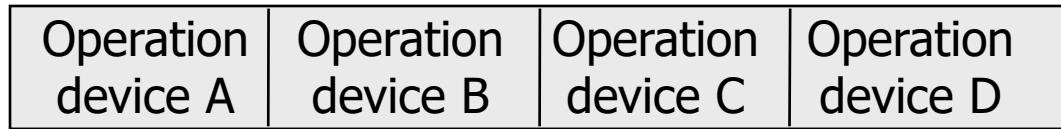
Upward calls are allowed (as long as there are no cycles).



Layer i+1

Layer i

Layer i-1

Finer resolution: Kernel



Processes

Kernel interface

Kernel operations

Process state change operations

Data structure operations

Kernel memory management          Dynamic systems only

# Types of realization of the kernel

- Scattered across programs

Program address spaces

Kernel operations
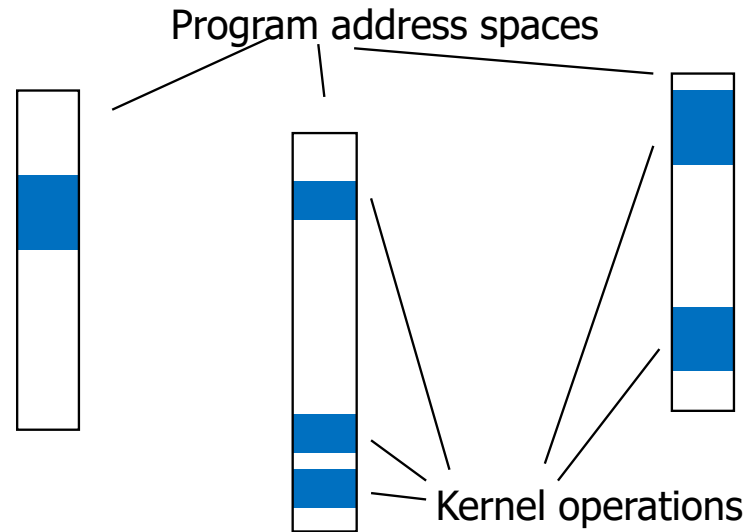
- Resident, compact, sealed

Kernel Address space

Kernel programs (Procedures)

Real (physical) memory

Processes (OS-processes, User processes)
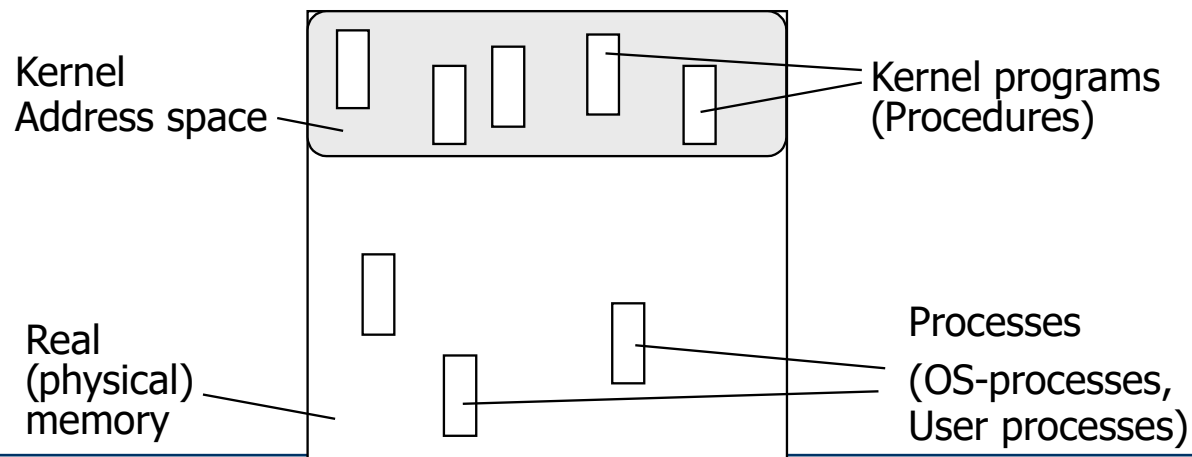
# Microkernel architecture

- The architecture just described is called microkernel architecture in the literature.

- Historically, people simply wanted to make a distinction to UNIX, in which the whole resource management (e.g. file system) is part of the kernel (macrokernel).

- Therefore the sizes of kernels differ widely.

- *Mach* (OSF-1) uses several MByte memory footprint, while *Cosy* (own development) only needs some 100KB.

- Such deviations in size sometimes lead to names like *Nanokernel* or *Picokernel*.

- There is no general agreement what should be in the kernel.

- However, process management and process communication are essential.

# Monolithical Systems
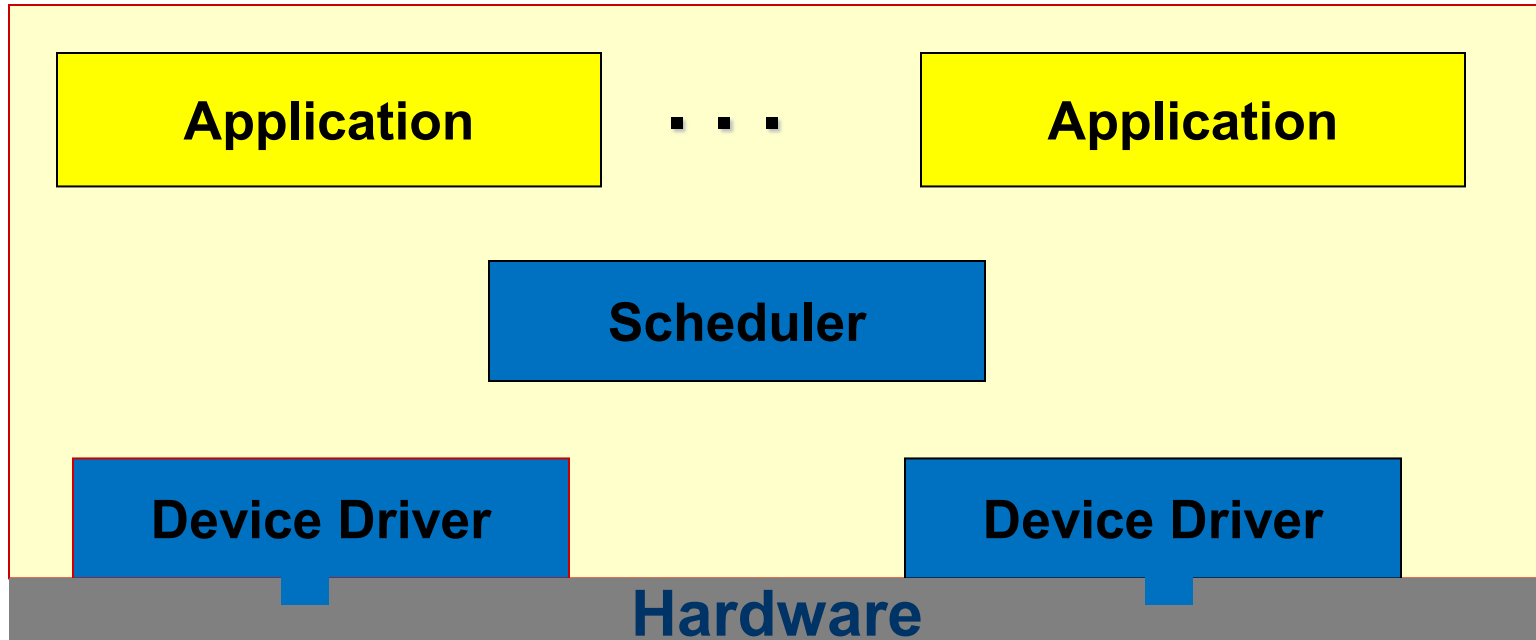


Besides microkernel OS, there are other approaches available:

In **monolithical systems** there is no strict separation between application and operating system.
Appropriate for small static OS, e.g. in embedded systems

# Monolithical OS-Kernel
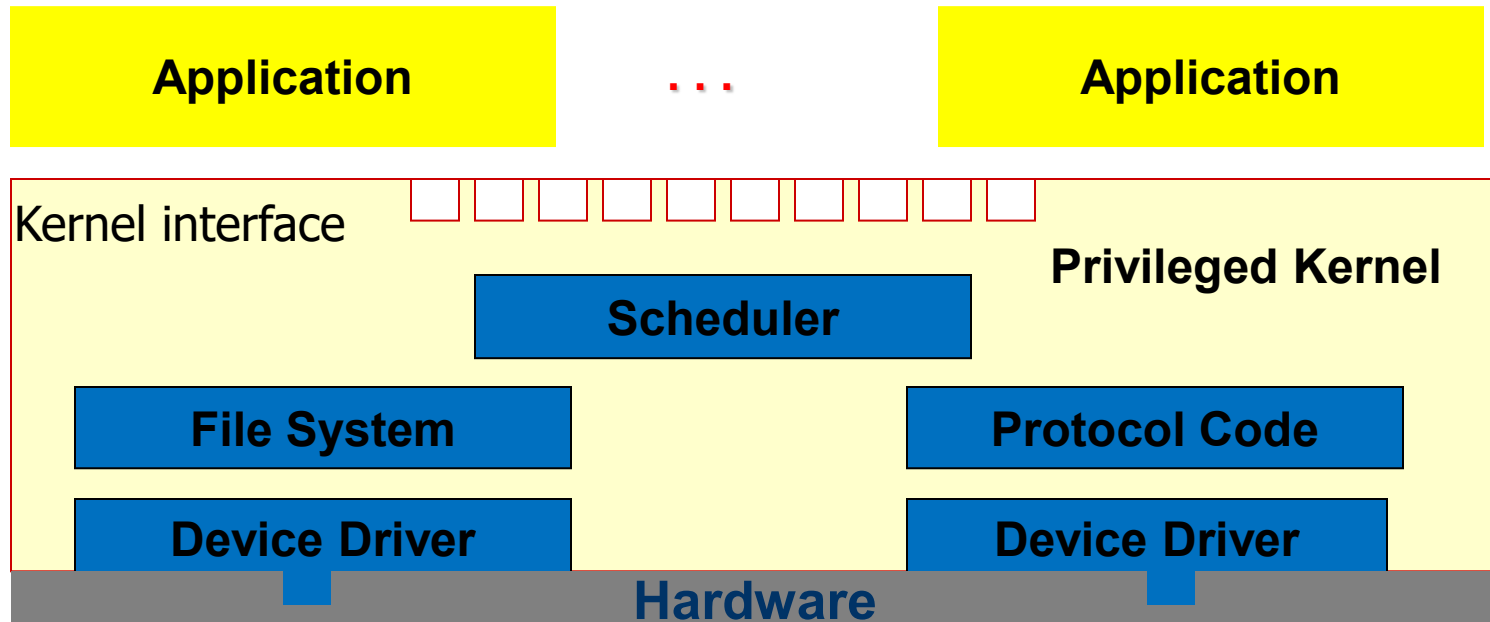


**Monolithical** OS-**kernels** do have a separation (protection) between application and OS, but no separation among OS components.

The whole kernel needs to be trustworthy.

# Microkernel OS
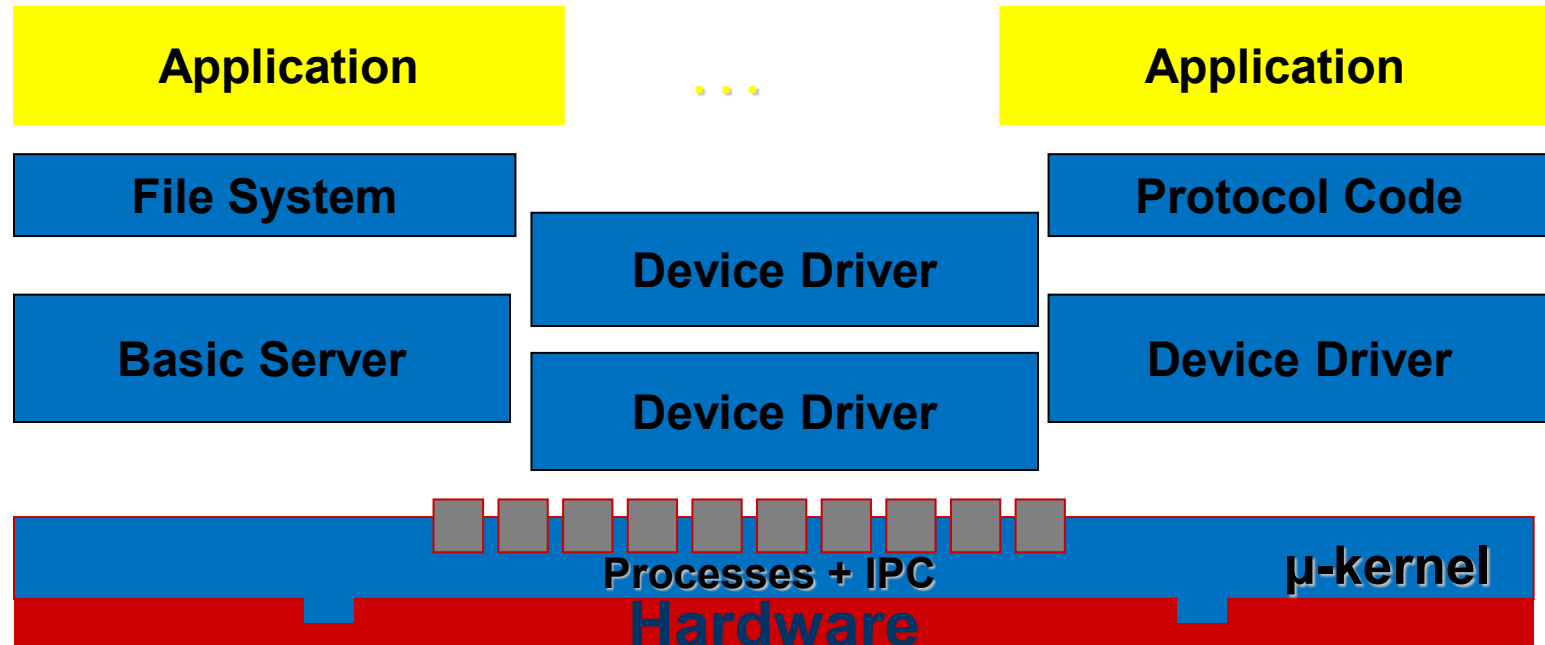


The kernel comprises process management (initializing, dispatching) and interprocess communication only.

- Clear kernel interface supports modular structure.

- Realization of services outside the kernel:

  - leads to more security and stability since the kernel will not be affected by faulty services,

  - improves flexibility and extendibility since services can be added and removed arbitrarily, even during operation.

- The safety-critical part of the system (kernel) is relatively small and can be verified easier.

- Usually, only the kernel needs to run in privileged mode.

- Microkernel architecture allows the coexistence of several alternative interfaces between OS and applications.

# Drawbacks of Microkernel architecture

- Usually worse performance

## Why?

- **Interplay of components outside the kernel requires more interprocess communication and therefore more kernel calls.**

KISS (keep it small and simple; keep it simple, stupid)
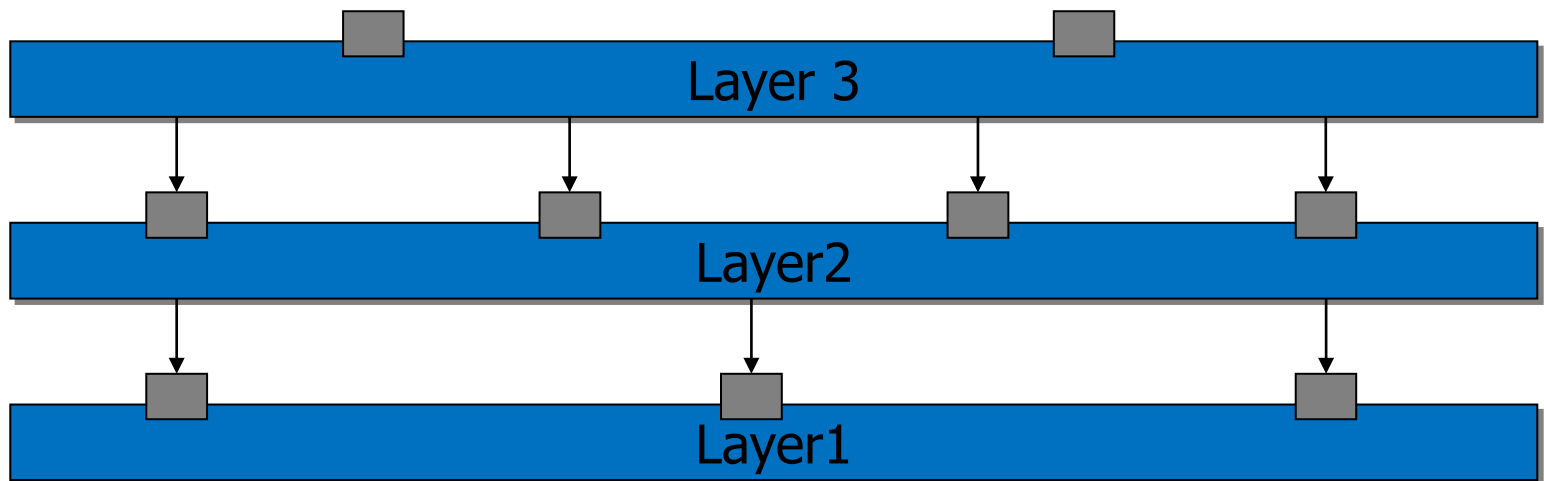
- Occam's Razor: "Plurality should not be assumed without necessity."
  (*William of Ockham*, *ca. 1285-1349*)

- "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
  (*C.A.R. Hoare, 1934-*)

- "Everything should be made as simple as possible, but not simpler." (*Albert Einstein, 1879-1955*)

- "Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away." (*Antoine de Saint-Exupéry, 1900-1944)*

# Modularization

- The system is decomposed into a set of modules such that
    - the interaction (information and control flows) within the module is high,
    - the interaction between modules is low,
    - the interfaces between the modules are simple,
    - the modules are easily understandable due to their limited size and complexity.

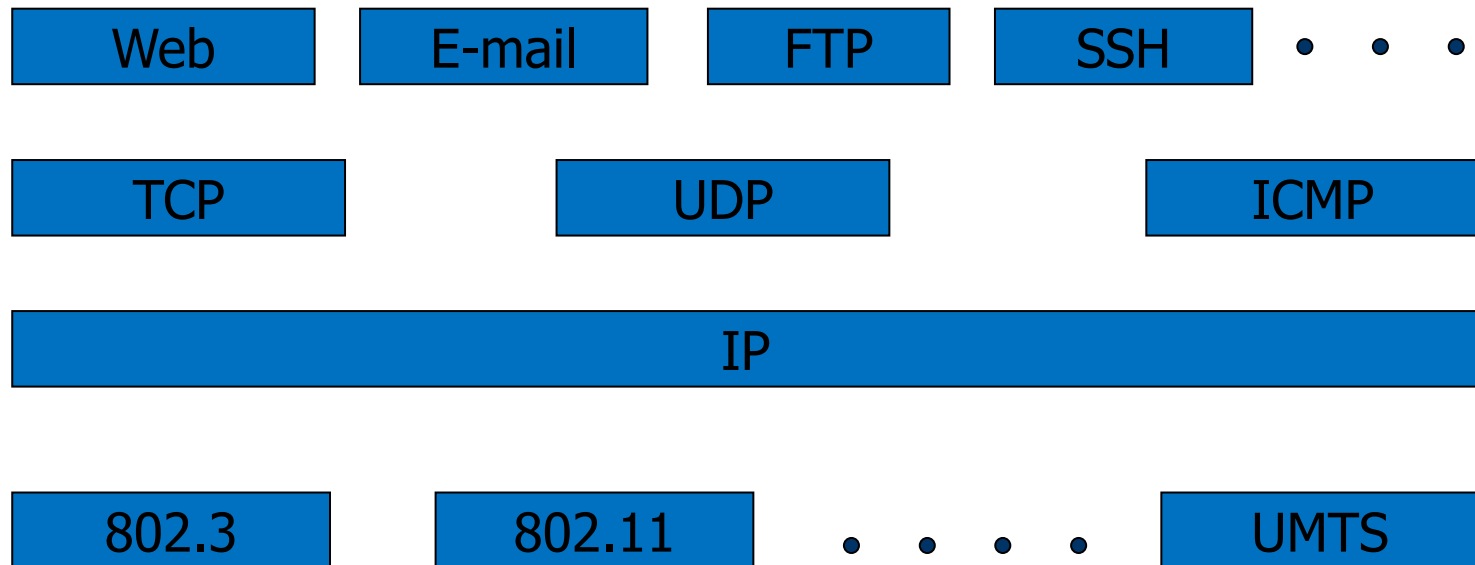- The principle can be applied hierarchically.

# Hierarchization

- Tree-like organization of homogeneous elements

- Goals:
  - scalability,
  - mastering complexity

# Layering
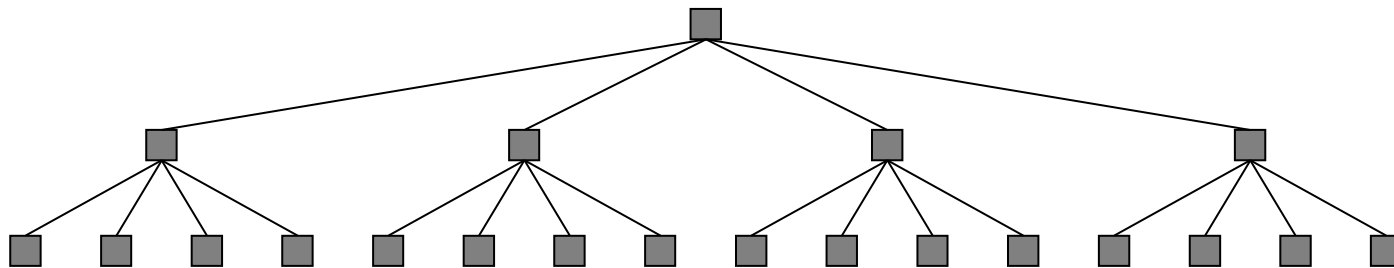
- Decomposition of a system's functionality into layers :
  - Simple, more universally usable functions more at the bottom
  - More complex and specific functions higher up
- Each layer represents an abstraction of lower layers.
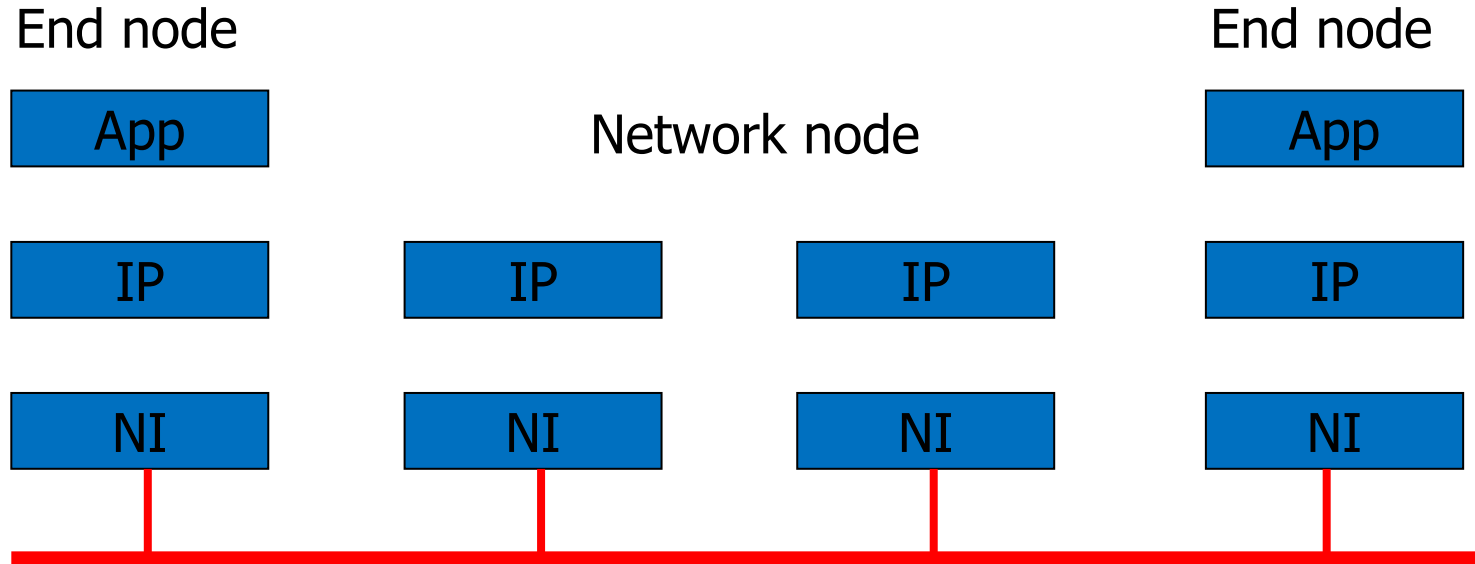- Each layer provides an interface that can be used by higher layers.

| Web | E-mail | FTP | SSH | • • • |
|-----|--------|-----|-----|-------|

| TCP | UDP | ICMP |
|-----|-----|------|

| IP |
|----|

| 802.3 | 802.11 | • • • • | UMTS |
|-------|--------|---------|------|

- End-to-end-Principle:

  „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.“

End node                                    End node

| App |          Network node          | App |

| IP |       | IP |       | IP |       | IP |

| NI |       | NI |       | NI |       | NI |

# Hourglass-Architecture (Internet)

Various Applications
(file transfer, media
streaming, web, email, VoIP)

**Internet protocol (IP)**

**Various networks, media,
and signal representations**

Diversity

Uniformity

Diversity
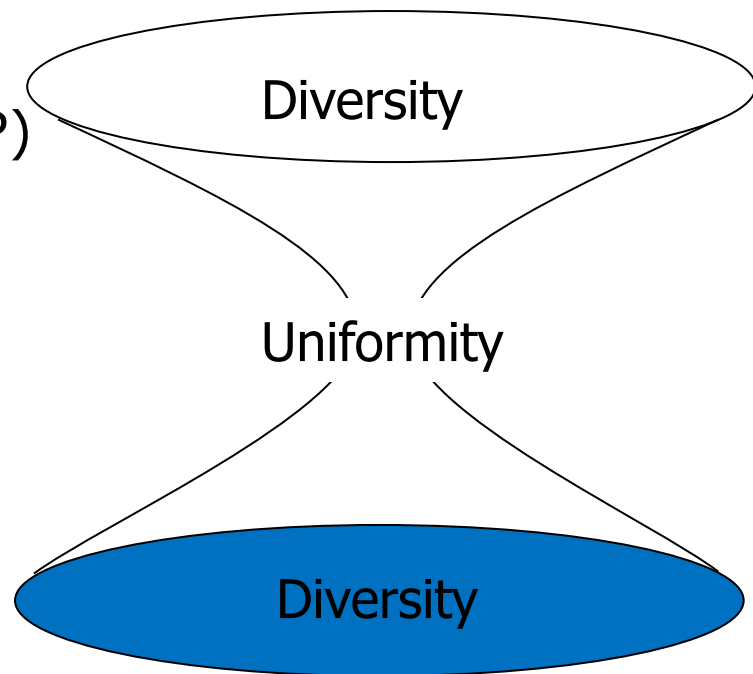
Various applications

**OS programming interface**

**Various Hardware**

Diversity

OS Interface

Diversity
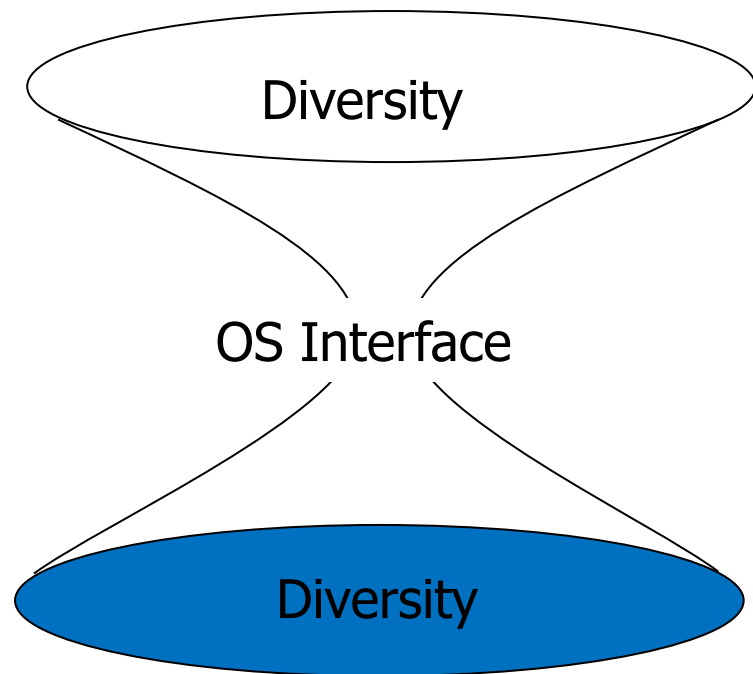
- A stable, universal programming interface should be provided.

- OS for a universal computer should be application neutral.

- Support for specific requirements should be placed in the topmost layer possible.

# Separation of policy and mechanism

- Application neutrality sometimes requires compromises:
  Lower layers may provide mechanisms that can be parameterized in higher layers to suit specific application requirements.

- Examples:
  - Scheduling
  - Paging
  - Security

# Orthogonality

- Functions and concepts of an OS should be independent of each other.

- Each component should exhibit orthogonal design criteria.

- Orthogonality means freedom of combination.
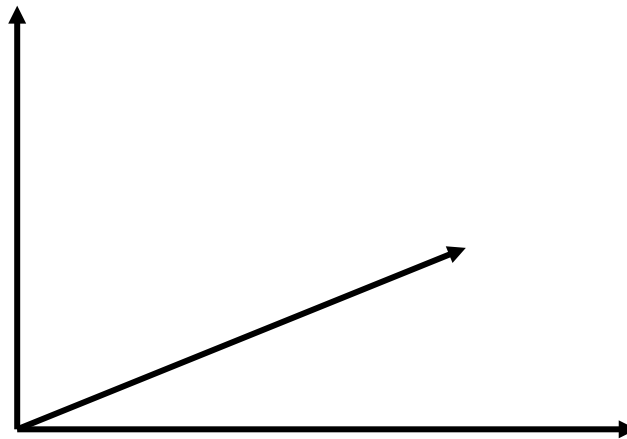
# SPOT-Rule (Single Point of Truth)

- No copies or repetitions
- For code: each functionality is implemented exactly once.
- For data: each piece of information that is managed by the system has exactly one representation.

- Usage of SPOT avoids inconsistencies.

# Further reading

- Liedtke, J.:     *Toward Real Microkernels*,
    Communications of the ACM 39,9 (Sept. 1996)

- Lampson, B.: *Hints for computer system design*,
    ACM Operating Systems Rev. 15, 5 (Oct. 1983),
    pp 33-48. Reprinted in *IEEE Software* 1, 1 (Jan. 1984),
    pp 11-28.

- Saltzer, J.; Reed, D.; Clark, D.:
    *End-to-End Arguments in System Design*,
    ACM Transactions on Computer Systems 2, 4
    (Nov. 1984) pp. 277-288.

    http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf