TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Operating Systems Group

DISTRIBUTED OPERATING SYSTEMS

OPERATING-SYSTEM ARCHITECTURES

https://tud.de/inf/os/studium/vorlesungen/dos

HORST SCHIRMEIER

Agenda

- SW Architecture: Terms and Differentiation

- Library Operating Systems
- Monolithic Systems
- Microkernels
- Exokernels and Virtualization

- Conclusion

Literature

Silberschatz, Chap. 23, „Influential Operating Systems“

Tanenbaum, Chap. 1.7, „Operating System Structure“

Agenda

- **SW Architecture: Terms and Differentiation**

- Library Operating Systems
- Monolithic Systems
- Microkernels
- Exokernels and Virtualization

- Conclusion

Literature

Silberschatz, Chap. 23, „Influential Operating Systems“

Tanenbaum, Chap. 1.7, „Operating System Structure“

Software Architecture

- Definition:

> *The basic organization of a system, represented by its* ***components****, their* ***relationships*** *to each other and to the environment, and the* ***principles*** *that determine the design and evolution of the system.*

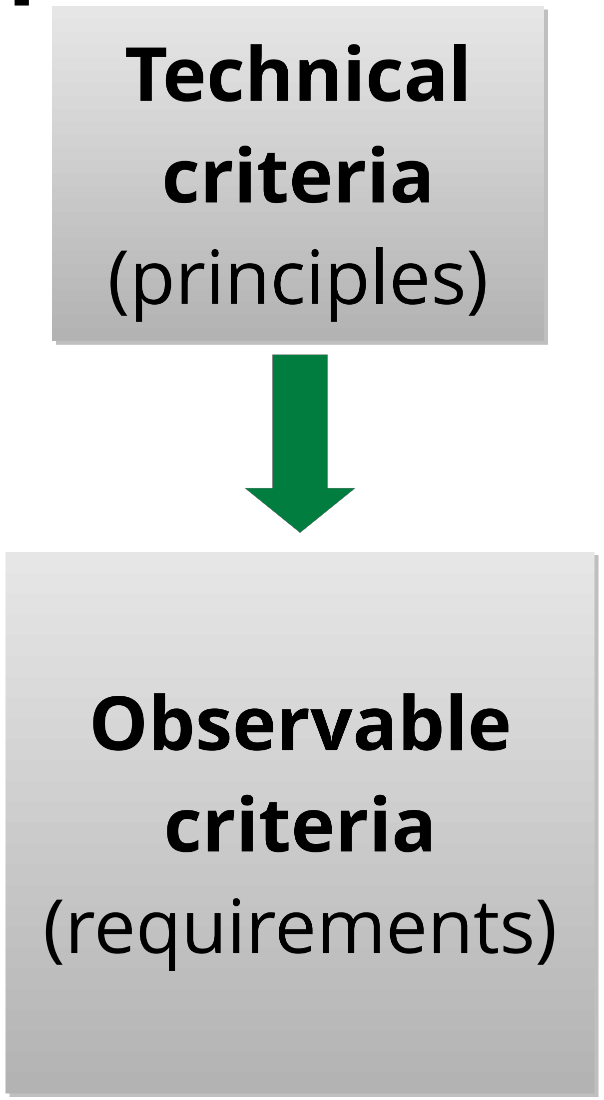
Translated from: Gesellschaft für Informatik e.V., https://gi.de/informatiklexikon/software-architektur

- Intuitively: “Boxes and arrows”
- Does *not* describe any design details
- … but connects **requirements** and a to-be-constructed **system**.

OS Architectures: Differentiation

- **Isolation**
- **Interaction mechanisms**
- **Interrupt-handling mechanisms**

- **Adaptability**
  - Porting, changes
- **Extensibility**
  - New functionality or services
- **Robustness**
  - Behavior in case of errors
- **Performance**

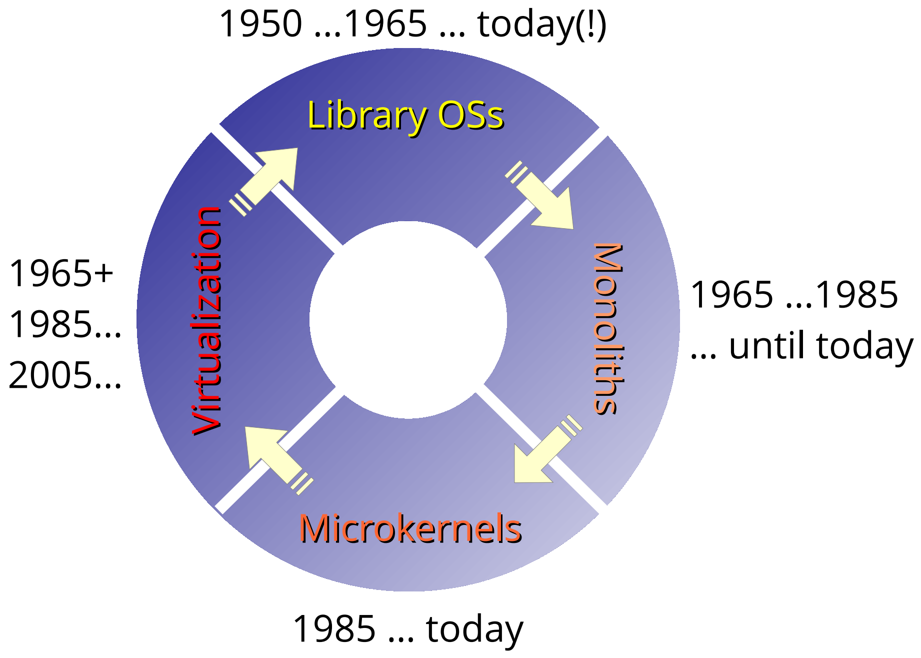


Agenda

- SW Architecture: Terms and Differentiation

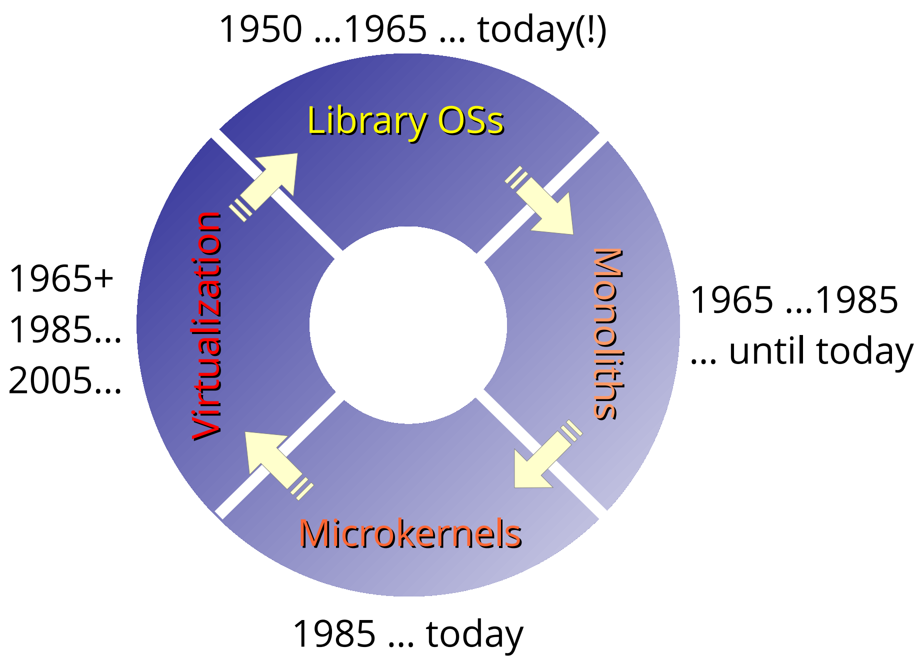
- Library Operating Systems
- Monolithic Systems
- Microkernels
- Exokernels and Virtualization

- Conclusion



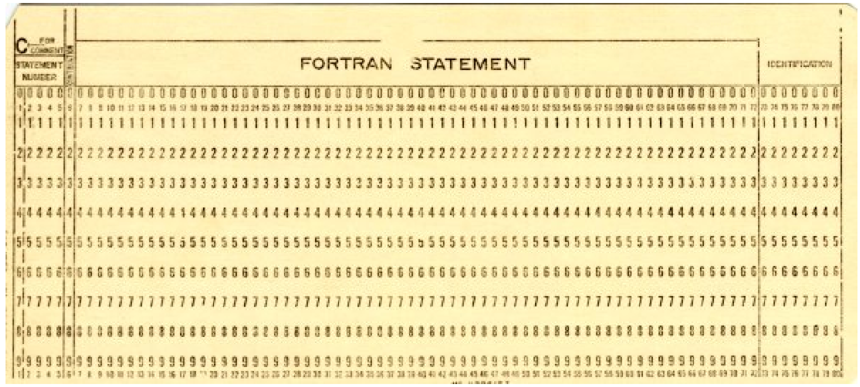
Agenda

- SW Architecture: Terms and Differentiation
- **Library Operating Systems**
- Monolithic Systems
- Microkernels
- Exokernels and Virtualization
- Conclusion

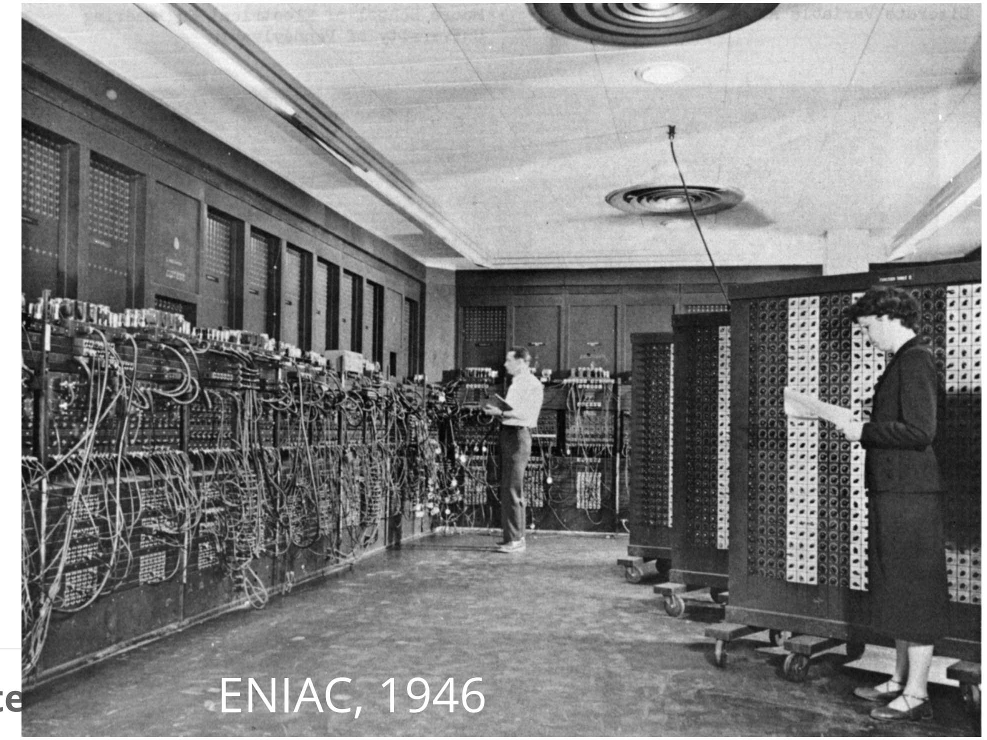


How Operating Systems Came to Be

- First computers ran without system software
  - **Every program** had to control the entire hardware by itself
  - Systems ran in batch mode, controlled by human operators
    - Single tasking, punch cards
  - Comparably simple periphery
    - Serially connected tape drives, punch-card readers/writers, printers
- **Duplication of device-programming code** in every application
  - Waste of developer + compile time, and of storage memory
  - Error-prone



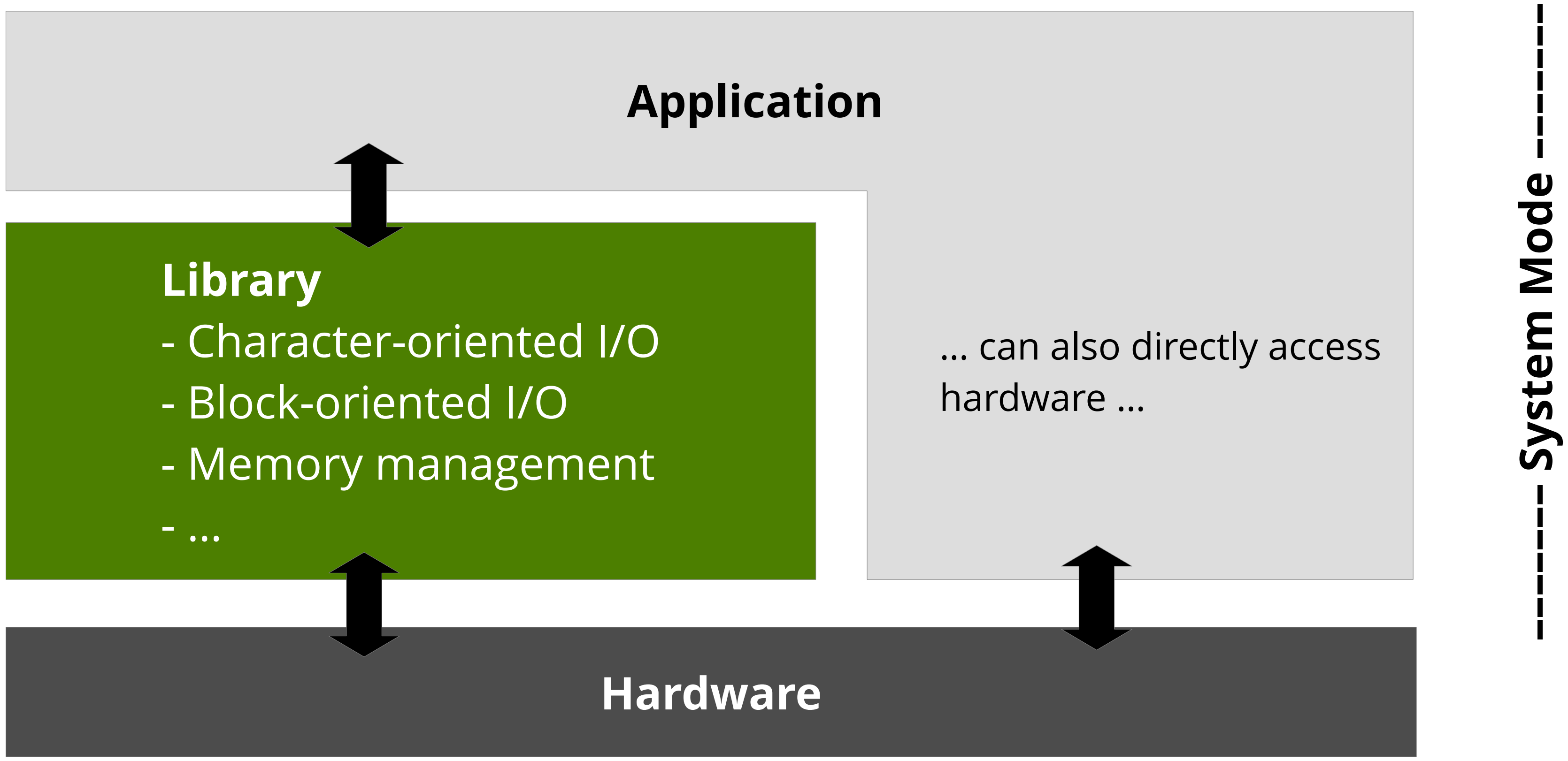
Punch card


ENIAC, 1946

Library Operating Systems

- Collection of often used device-programming functions in **software libraries**
  - ➔ Could be used by all programs
  - ➔ “System calls” as normal **function calls**
- Library could stay in machine’s memory
  - ➔ Lower program load times, **“resident monitor”**
- Library functions were **documented and tested**
  - ➔ Lower development effort for application programmers
- **Centralized bug fixing**
  - ➔ Better reliability

Library Operating Systems



Library OS: Evaluation

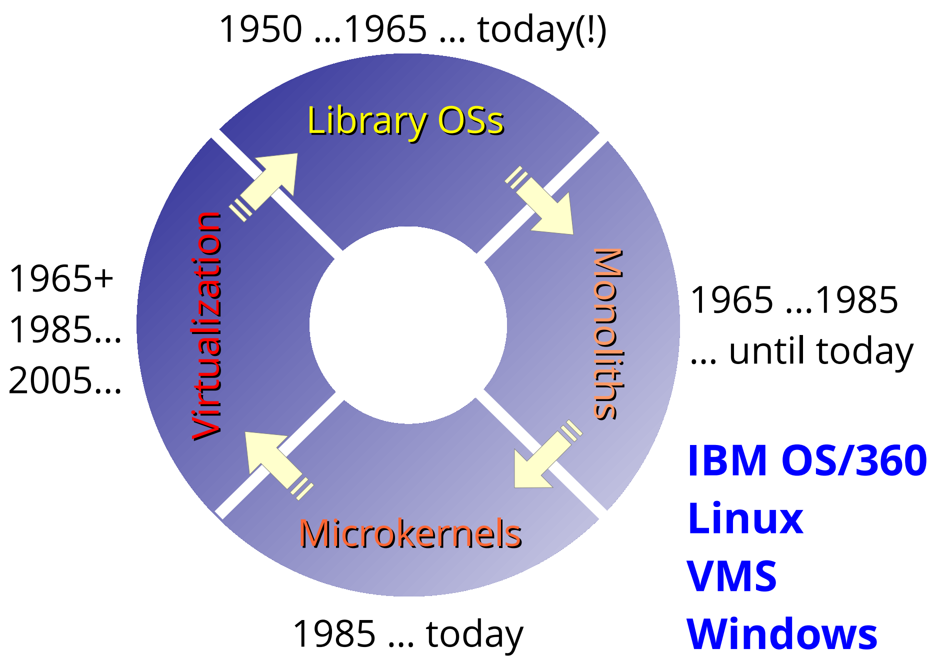
- **Isolation**
  - Ideal: ***Single Tasking*** system – but with high “task-switching times“
- **Interaction mechanisms**
  - Direct (function calls)
- **Interrupt-handling mechanisms**
  - Partially no interrupts at all → ***Polling***
- **Adaptability**
  - Own library for each hardware architecture, no standardization
- **Extensibility**
  - Depending on library structure: Global structures, **“spaghetti code”**
- **Robustness**
  - Direct control over entire hardware: Error → system halts
- **Performance**
  - Very high – direct operations on hardware w/o privilege-separation mechanisms

Library OS: Discussion

- Productive use of expensive computer hardware only at **low fraction of time**
  - High time effort for switching to the next application
  - I/O wait unnecessarily wastes time (only one process on the system)
- Long waiting times for results
  - Waiting queue, batch processing
- No interactivity
  - System run by operators, no direct access to hardware
  - Programs not accessible at runtime

Agenda

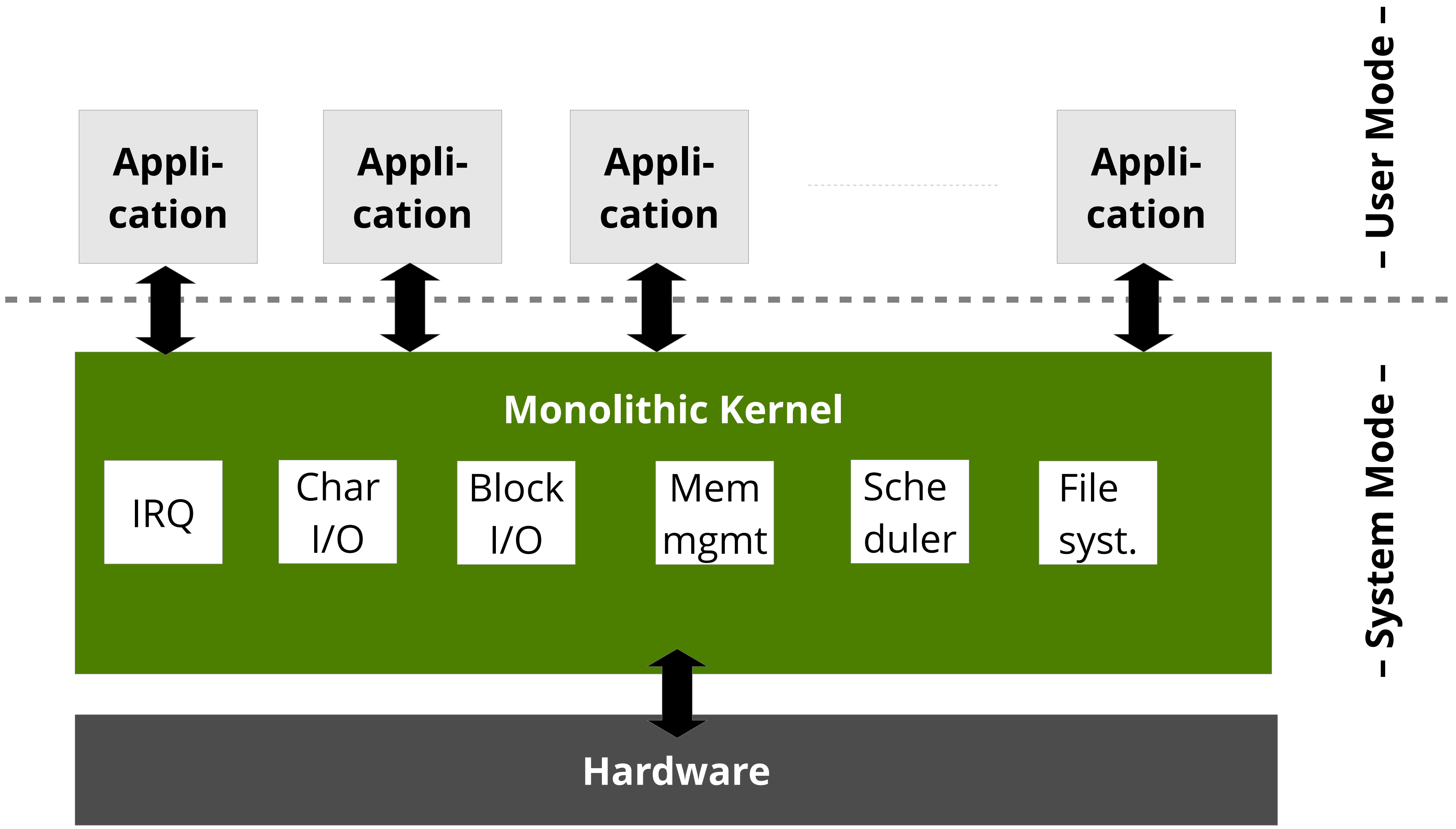
- SW Architecture: Terms and Differentiation
- Library Operating Systems
- **Monolithic Systems**
- Microkernels
- Exokernels and Virtualization
- Conclusion



Monolithic Operating Systems

- Administration of computer hardware
  - Standardized **accounting** of system resources
- **Complete control** of hardware and software
  - Applications now run controlled by the OS
  - Multi-process systems became possible: ***Multiprogramming***
- Introduction of a privilege system
  - System mode, user mode – with hardware support
  - Direct hardware access only in system mode
- System calls with special mechanisms (***Software Traps***)
  - Necessitate context save and -switch

Monolithic Operating Systems



Monolithic Systems: OS/360

- One of the first monolithic OSs: IBM OS/360, 1966
- Goal: Common batch OS for all IBM mainframes
  - But – performance and memory varied by orders of magnitude!
- Availability in different configurations:
  - **PCP** (*Primary Control Program*): Single-process, small systems
  - **MFT** (*Multiprogramming with Fixed number of Tasks*): medium-sized systems (256 kiB RAM!), fixed memory partitioning for processes, fixed number of tasks
  - **MVT** (*Multiprogramming with Variable number of Tasks*): high-end, swapping, optional *Time Sharing Option* (TSO) for interactive use
- Trend-setting features:
  - Hierarchical file system
  - Processes could spawn subprocesses
  - MFT and MVT are API and ABI compatible

IBM's z/OS still supports OS/360 applications today

Monolithic Systems: OS/360

- Fred Brooks' book "*The Mythical Man-Month*" describes organizational and technical problems in the development process
  - **Conceptual Integrity**
    - **Separation of architecture and implementation** was hard to achieve. Developers tend to cram in every technically possible feature the users ask for.
      → Reduces user-friendlyness and understandability, and derails developer productivity
  - ***"Second System Effect"***
    - Developers tried to remedy all shortcomings of the previous OS and to add all missing features.
      → Over-engineering, does not get finished
  - Too **complex dependencies** between system components
    - With a certain system size, the number of errors becomes irreducible.
- Progress in software engineering was driven by OS development

Monolithic Systems: Unix

- Developed as an OS for machines with little to moderate resources (Bell Labs)
  - Kernel size in 1979 (*7th Edition Unix*, PDP11):
    ~10,000 lines of code (manageable!), compiled ca. 50 KiB
  - Originally written by 2–3 developers
- Introduction of simple abstractions
  - Every system object can be represented as a **file** – a simple, unformatted stream of bytes
  - Complex functionality can be achieved by combining **simple system tools** (***Shell Pipelines***)
- New goal: **Portability**
  - Easy adaptability to different hardware
  - Development in "C" – designed as a domain-specific language for OS development

Monolithic Systems: Unix

- Further development
  - Systems with large address spaces (VAX; RISC systems)
  - Kernel continuously grew (System III, System V, BSD) – w/o substantial structural changes
  - Integration of highly complex subsystems
    - TCP/IP was about the same size as the rest of the kernel
- Linux development oriented itself along the structure of System V Unix
- Influence in academia: **"Open Source"** policy of *Bell Labs*
  - Unix shortcomings led to new research approaches
  - Many projects (e.g. Mach) tried to stay **compatible**

Monolithic OS: Evaluation

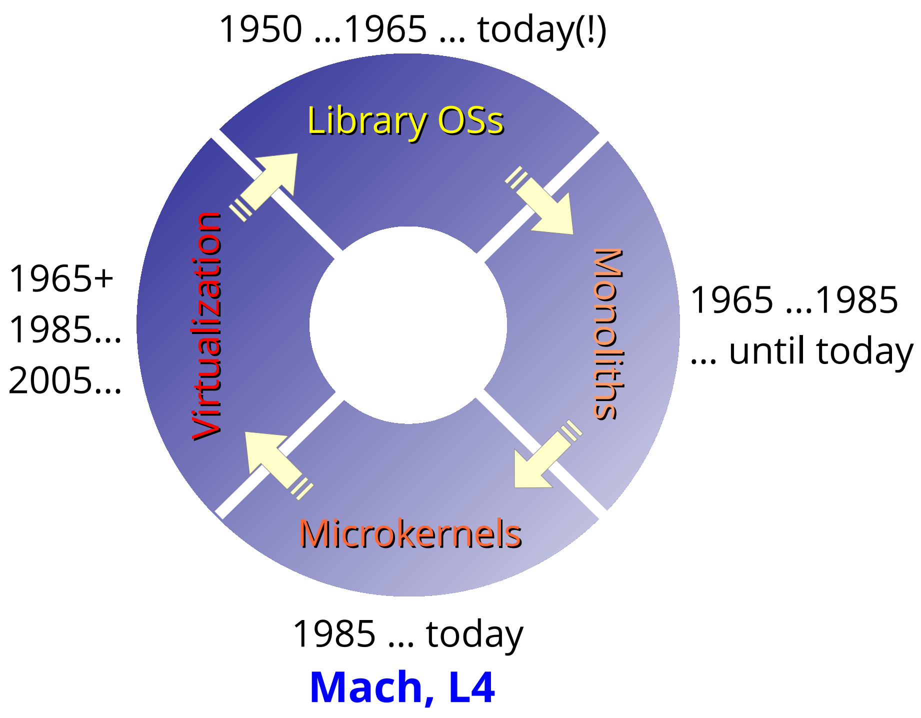
- **Isolation**
  - No isolation of kernel components, only between application processes
- **Interaction mechanisms**
  - Direct function calls (in the kernel), traps (application → kernel)
- **Interrupt-handling mechanisms**
  - Direct handling of hardware interrupts in interrupt handlers
- **Adaptability**
  - Changes to one component influences other components
- **Extensibility**
  - Originally: recompilation necessary; later: module system
- **Robustness**
  - Low – error in one component affects whole system
- **Performance**
  - High – little copying necessary, since all kernel components work in the same address space. However, system calls require a trap.

Monolithic OS: Discussion

- Complex monolithic kernels are hard to maintain
  - Adding or changing functionality often affects more modules than intended
- Shared address space
  - Security problem in one component (e.g. **Buffer Overflow**) compromises whole system
  - Many components unnecessarily run in system mode
- Limitations by coarse-grained synchronization
  - Often only a ***"Big Kernel Lock"***, i.e. only one process can run in kernel mode at one time, all others wait
  - Particularly a performance-reducing factor in multiprocessor systems

Agenda

- SW Architecture: Terms and Differentiation
- Library Operating Systems
- Monolithic Systems
- **Microkernels**
- Exokernels and Virtualization
- Conclusion



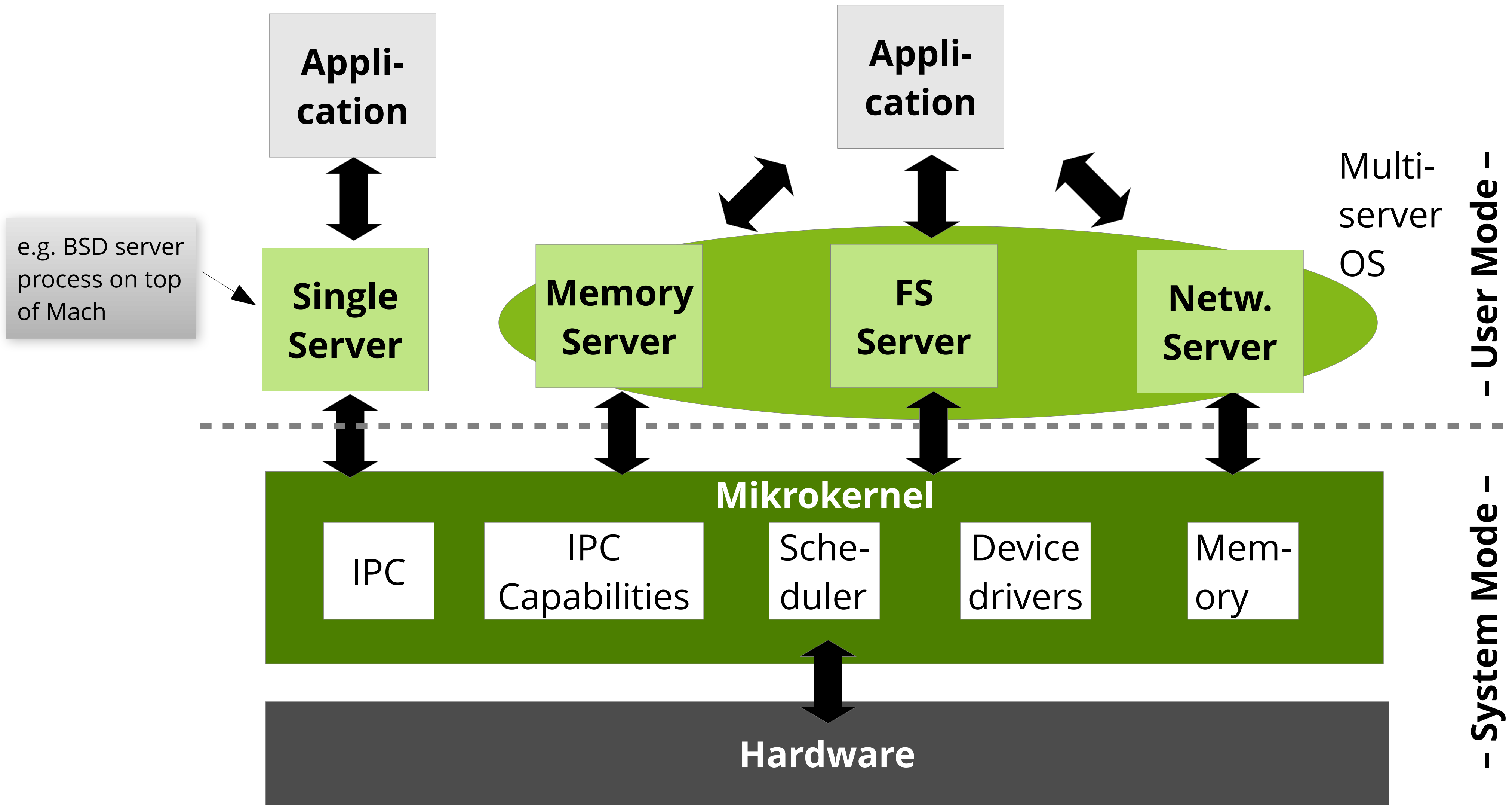
Microkernel Systems

- Goal: Size reduction of the ***Trusted Computing Base***
  - Minimize functionality implemented in CPU's system mode
  - Isolate remaining components in non-privileged user mode
- Principle of Least Privilege (POLP)
  - System services only get privileges that are absolute necessary to fulfill their task.
- Invocation of system services and process-to-process communication via messages (IPC – ***Inter-Process Communication***)
- Reduced functionality in the microkernel
  - Smaller code size (10,000 lines of C++ code vs. ~16 million lines of C in Linux w/o device drivers)
  - Puts formal verification approaches within reach (seL4)

1st Generation Microkernels

- **Example**: CMU Mach
- Starting point: Separating BSD Unix features in
  - those requiring system mode, and
  - those that don’t.
- Goal: Extremely **portable system**
- Improved Unix concepts:
  - New communication mechanisms – IPC and **Ports**
    - Ports: protected IPC channels
    - IPC optionally network transparent – support for distributed systems
  - Parallel activities within a process address space
    - Support for **threads** – a process is now a framework for one or more threads
    - Improved support for multiprocessor systems

1st Generation Microkernels



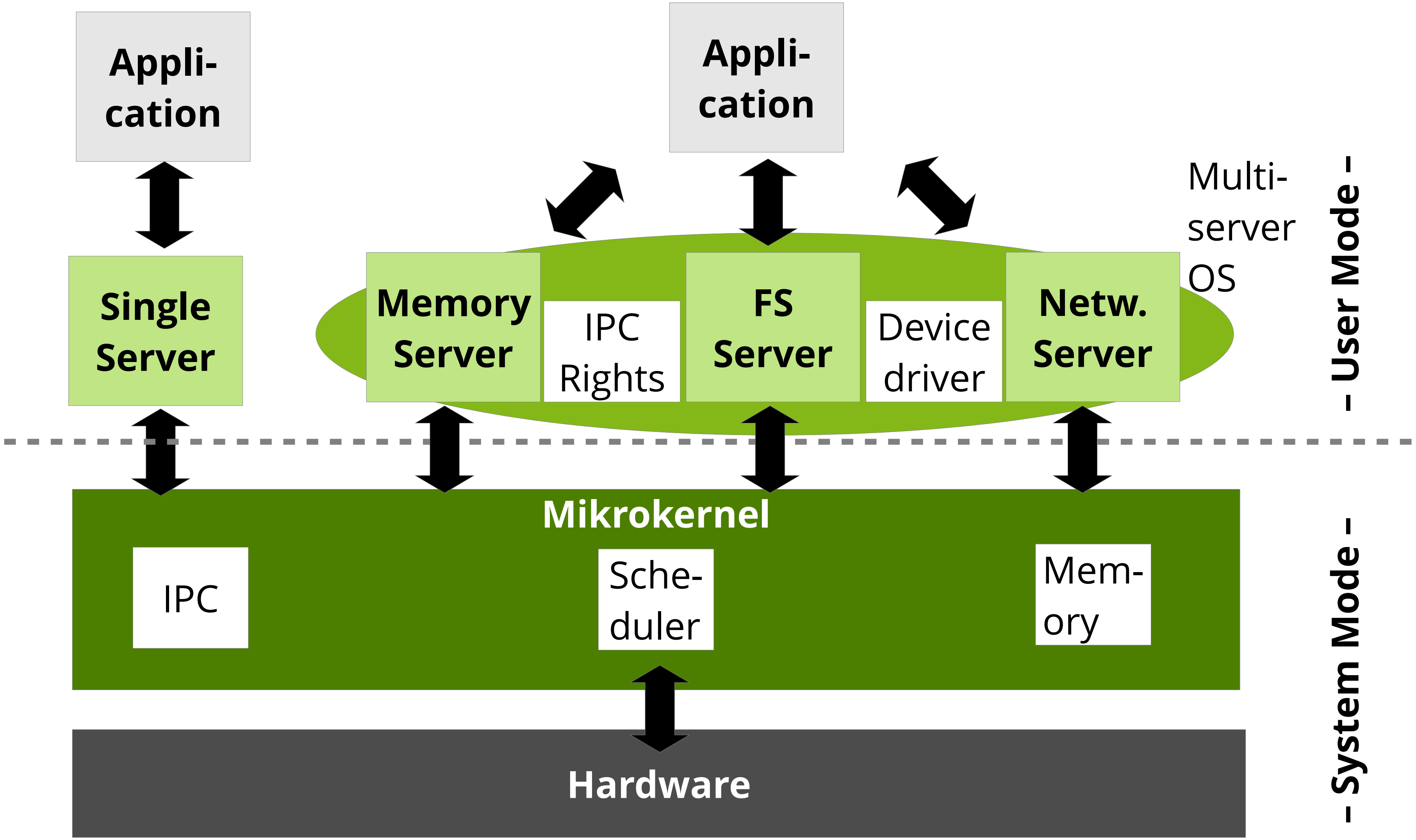
1st Generation Microkernels

- Mach shortcomings:
  - high IPC overhead
    - **System calls 10x slower** compared to monolithic kernels
  - Suboptimal decisions which components to implement in the kernel: large code base
    - Device drivers and IPC / Port access rights management in the kernel
  - Result: **Generally bad reputation of microkernels**
    - Widespread doubt regarding **practical usability**
- Microkernel idea was **dead in the mid 1990s**
- Practical use of Mach mostly in hybrid systems
  - Separately developed components for microkernel and servers
  - Colocation of components in a single address space, replacement of in-kernel IPC by function calls
  - **Apple MacOS X**: Mach 3 microkernel + FreeBSD

2nd Generation Microkernels

- Goal: Address shortcomings of 1st generation
  - Faster IPC operations
  - Jochen Liedtke: L4 (1996)
    - *A* ***concept is tolerated inside the microkernel*** *only if moving it outside the kernel, i.e., permitting competing implementations, would* ***prevent the implementation*** *of the system's required functionality.*
- 4 basic mechanisms:
  - Abstraction of **address space**
  - **Thread** model
  - **Synchronous communication** between threads
  - **Scheduling**
- A lot of functionality was pushed out of the microkernel to user space
  - e.g. IPC rights checking

2nd Generation Microkernels

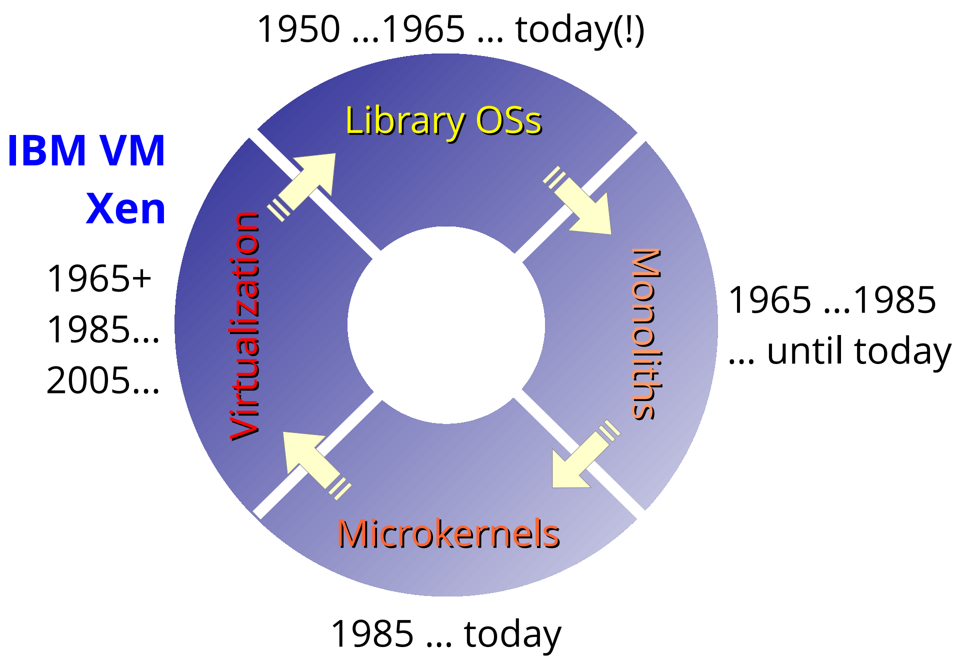


Microkernel OS: Evaluation

- **Isolation**
  - Very good – separate address space for each component
- **Interaction mechanisms**
  - Synchronous IPC
- **Interrupt-handling mechanisms**
  - Translation of interrupts to IPC messages in the microkernel
- **Adaptability**
  - Originally hard to adapt / port – x86 assembler, later reimplementations in C/C++
- **Extensibility**
  - Very good and simple, components in user mode
- **Robustness**
  - Good – but depends on robustness of servers
- **Performance**
  - Primarily dependent on IPC performance

Agenda

- SW Architecture: Terms and Differentiation
- Library Operating Systems
- Monolithic Systems
- Microkernels
- **Exokernels and Virtualization**
- Conclusion



Exokernels: Even smaller than µ-kernels

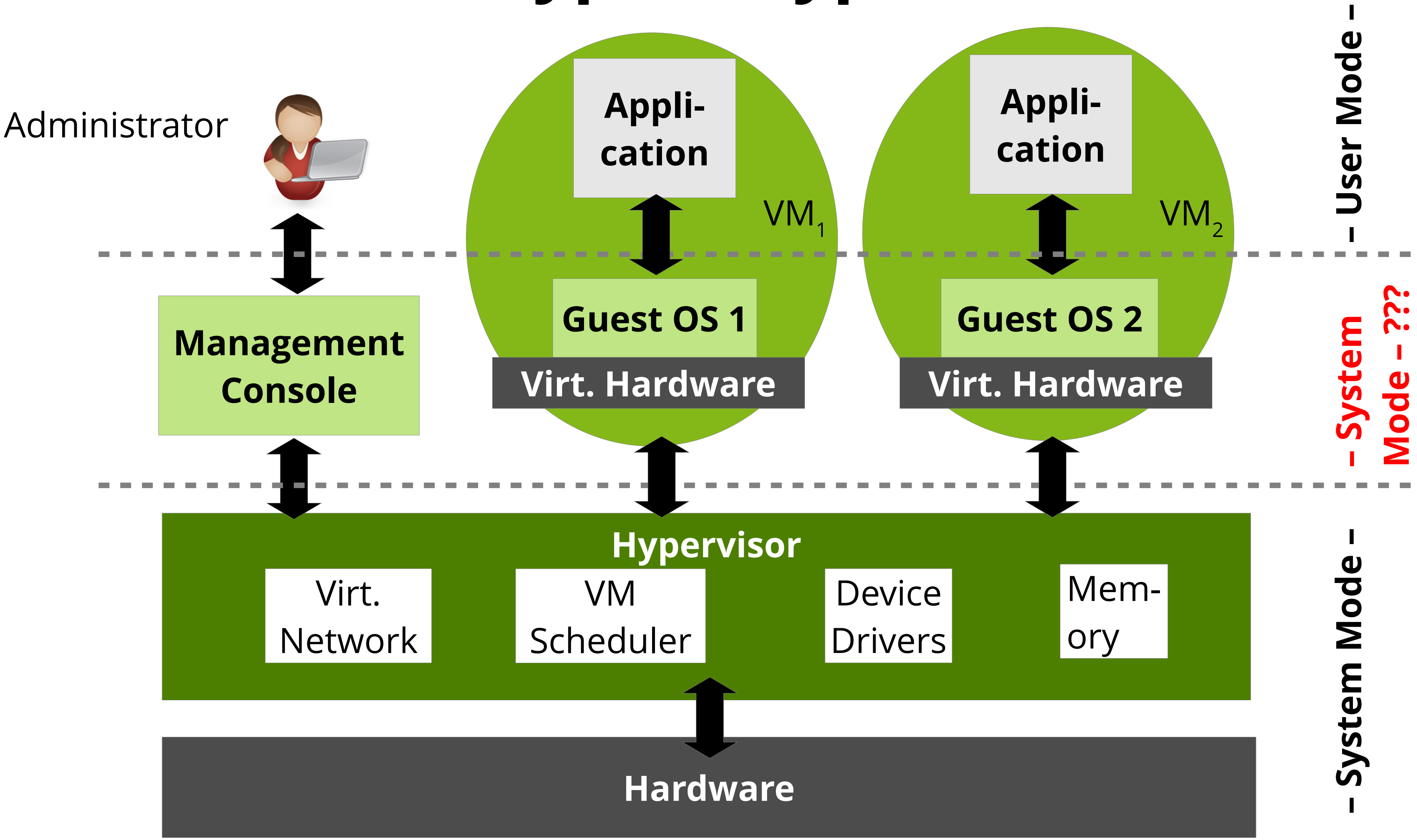
Basic simplification idea:

- Lowest system-software layer …
  - does not implement **strategies nor abstractions**,
  - does not **virtualize resources**
- Sole responsibility: **Resource partitioning**
  - Each application gets assigned own resources
  - Exokernel enforces partitioning
  - **Application-specific library OSs** implement everything else *inside* the resource container.
- Disadvantage: Library operating systems are **exokernel specific**

Virtualization

- Goal: Isolation and multiplexing of resources *below* the OS layer
  - Simultaneous use of multiple **guest operating systems**
- Virtual machines (VMs) on system level virtualize all hardware resources, incl.
  - CPUs, main memory, hard disks, peripheral devices
- ***Virtual Machine Monitor*** (VMM) or ***Hypervisor***:
  Software component that provides the virtual-machine abstraction

Virtualization (with Type-1 Hypervisor)

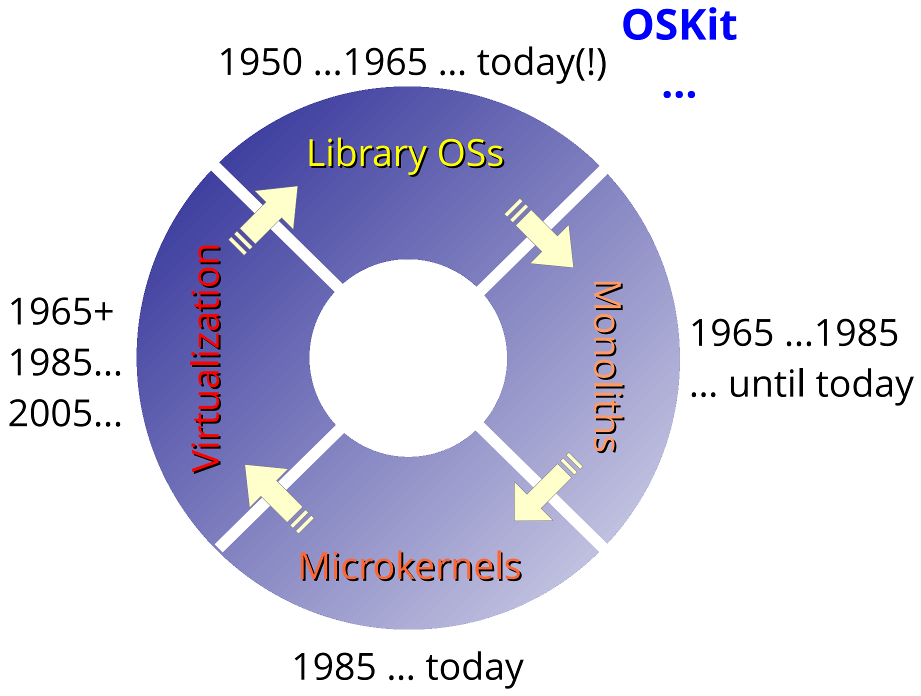


Virtualization: Evaluation

- **Isolation**
  - Very good – but coarse-grained (between VMs)
- **Interaction mechanisms**
  - Communication between VMs only via TCP/IP (virtual network interface cards!)
- **Interrupt-handling mechanisms**
  - Type-1 hypervisor / host OS forwards IRQs to guest kernel inside the VM (simulated HW interrupts)
- **Adaptability**
  - Implementation CPU-type specific (+ paravirtualization requires high effort)
- **Extensibility**
  - Hard – not available in most VMMs
- **Robustness**
  - Good – but coarse-grained (whole VMs affected by crashes)
- **Performance**
  - Good – 5–10% slowdown vs. execution on bare metal

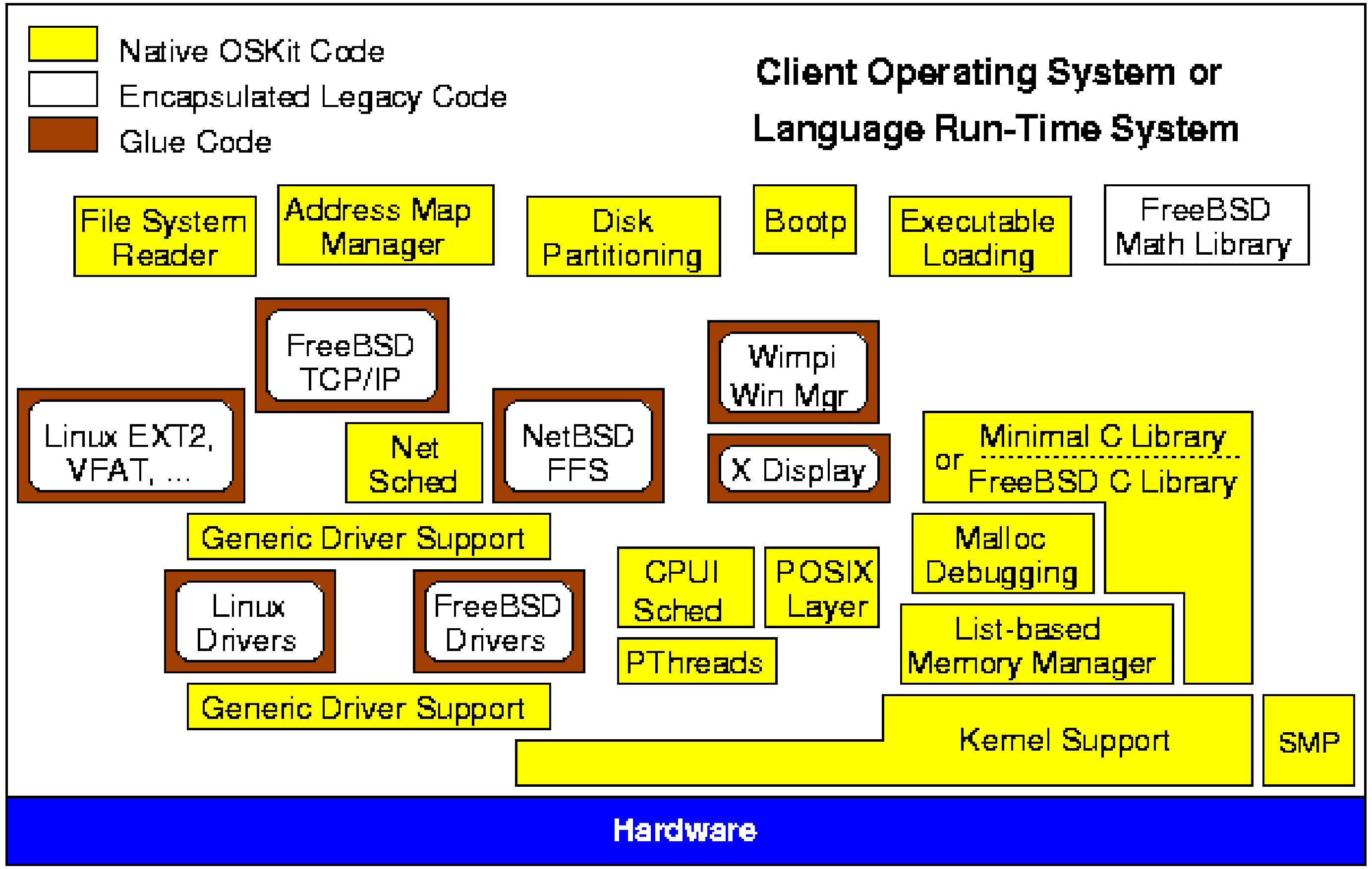
Agenda

- SW Architecture: Terms and Differentiation
- **Library Operating Systems**
- Monolithic Systems
- Microkernels
- Exokernels and Virtualization
- Conclusion



OS-Functionality Libraries

- ***"Unikernels"*** are meant for efficient execution of *one* application within a VM
  - mirageOS, Mini-OS, Unikraft, …
- Example: Utah OSKit
  - "Best Of" different OS components
  - Interfaces adapted to common standard
  - Language support (interface generator) simplifies integration



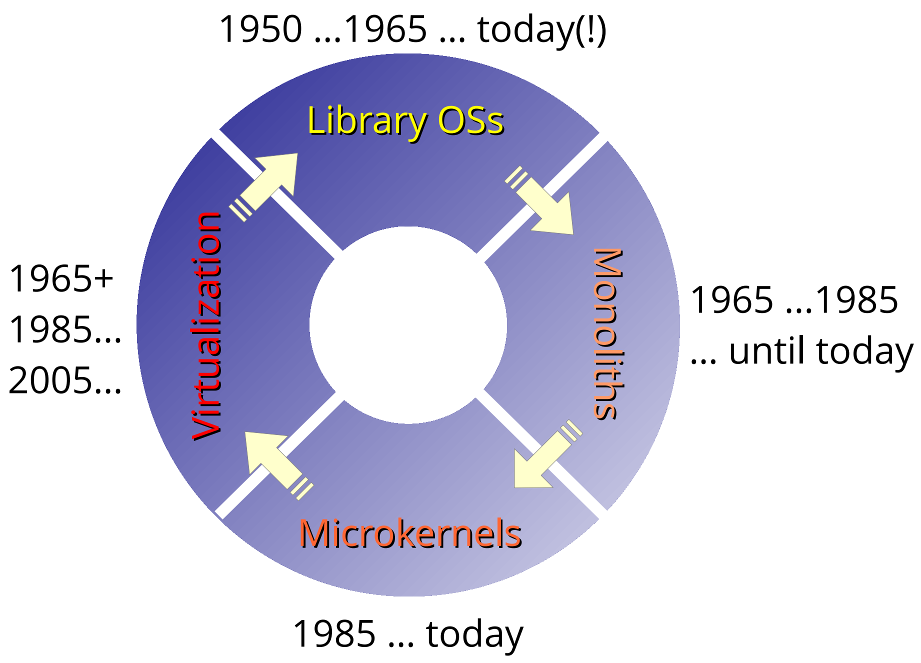
TECHNISCHE UNIVERSITÄT DRESDEN

Agenda

- SW Architecture: Terms and Differentiation

- Library Operating Systems
- Monolithic Systems
- Microkernels
- Exokernels and Virtualization

- **Conclusion**



Operating-System Architectures: Conclusion

- OS architectures are still an **active research topic**
  - “Old” technology like virtualization meet new application areas, e.g. in **cloud computing**
  - Hardware and applications continuously change, e.g.
    - Energy savings (energy harvesting)
    - Scalability (multi- / manycore processors)
    - Handling heterogeneity (Arm big.LITTLE, Intel Performance hybrid architecture, GPUs, …)
    - Adaptability (mobile systems, resource-constrained systems)
    - Persistent main memories (TI FRAM, Intel Optane)
- Slowdown of mainstream acceptance: **Compatibility requirements** and **high development cost**
  - Virtualization as a compatibility layer