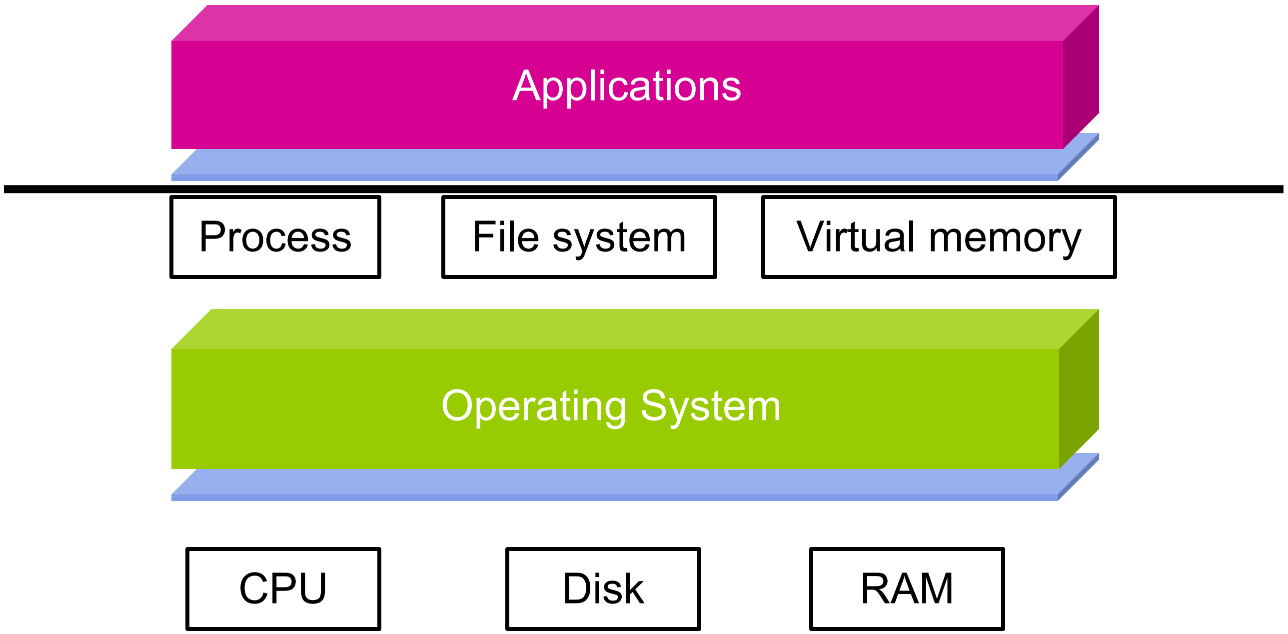
Extensible OS Design

CS 202: Advanced Operating Systems

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Recall: OS Organization



Separate OS and User space

Why is the structure of an OS important?

- Protection
  - User from user and system from user
- Performance
  - Does the structure facilitate good performance?
- Flexibility/Extensibility
  - Can we adapt the OS to the application
- Scalability
  - Performance goes up with more resources
- Agility
  - Adapt to application needs and resources
- Responsiveness
  - How quickly it reacts to external events
- Can it meet these requirements?

Extensibility

- What do we mean by extensibility?
  - Flexible to add new features/functionalities
  - Customization
  - Good efficiency
  - Good security

- Can you give a few examples?
  - Browser plugins/extensions
  - Device drivers
  - Virtual memory policy, OS scheduler, …

Simple extensions

Extensibility context

- Traditional OS provide standard
  - Fixed set of abstractions
    - Processes, threads, VM, Files, IPC
    - Reachable through syscalls
  - Resource allocation and management
  - Protection and security

- Industry complaining of OS large overheads
  - Research community started asking how to provide customization

Motivation

- OS does not perform well for specific applications
  - Applications are very different (e.g., video game vs. number crunching application)
  - Scheduling policies, memory management, file systems, ...

Motivation

- OS does not perform well for specific applications
  - Applications are very different (e.g., video game vs. number crunching application)
  - Scheduling policies, memory management, file systems, ...
- Traditional centralized resource management cannot be specialized, extended or replaced
  - Substantial advantages achievable from specializing resource allocation

Motivation

- OS does not perform well for specific applications
  - Applications are very different (e.g., video game vs. number crunching application)
  - Scheduling policies, memory management, file systems, ...
- Traditional centralized resource management cannot be specialized, extended or replaced
  - Substantial advantages achievable from specializing resource allocation
- Fixed high-level abstractions too costly for good efficiency

Motivation

- OS does not perform well for specific applications
  - Applications are very different (e.g., video game vs. number crunching application)
  - Scheduling policies, memory management, file systems, ...
- Traditional centralized resource management cannot be specialized, extended or replaced
  - Substantial advantages achievable from specializing resource allocation
- Fixed high-level abstractions too costly for good efficiency
- Privileged software must be safely used by all applications
  - But protection and management interfere with performance and flexibility

Cost of Protection

- How expensive are border crossings?

Cost of Protection

- How expensive are border crossings?
- Procedure call within the same address space:
  - Save some general-purpose registers and jump

Cost of Protection

- How expensive are border crossings?
- Procedure call within the same address space:
  - Save some general-purpose registers and jump
- Mode switch:
  - Trap or syscall overhead
  - Switch to kernel stack
  - Switch some registers
  - 100s of ns

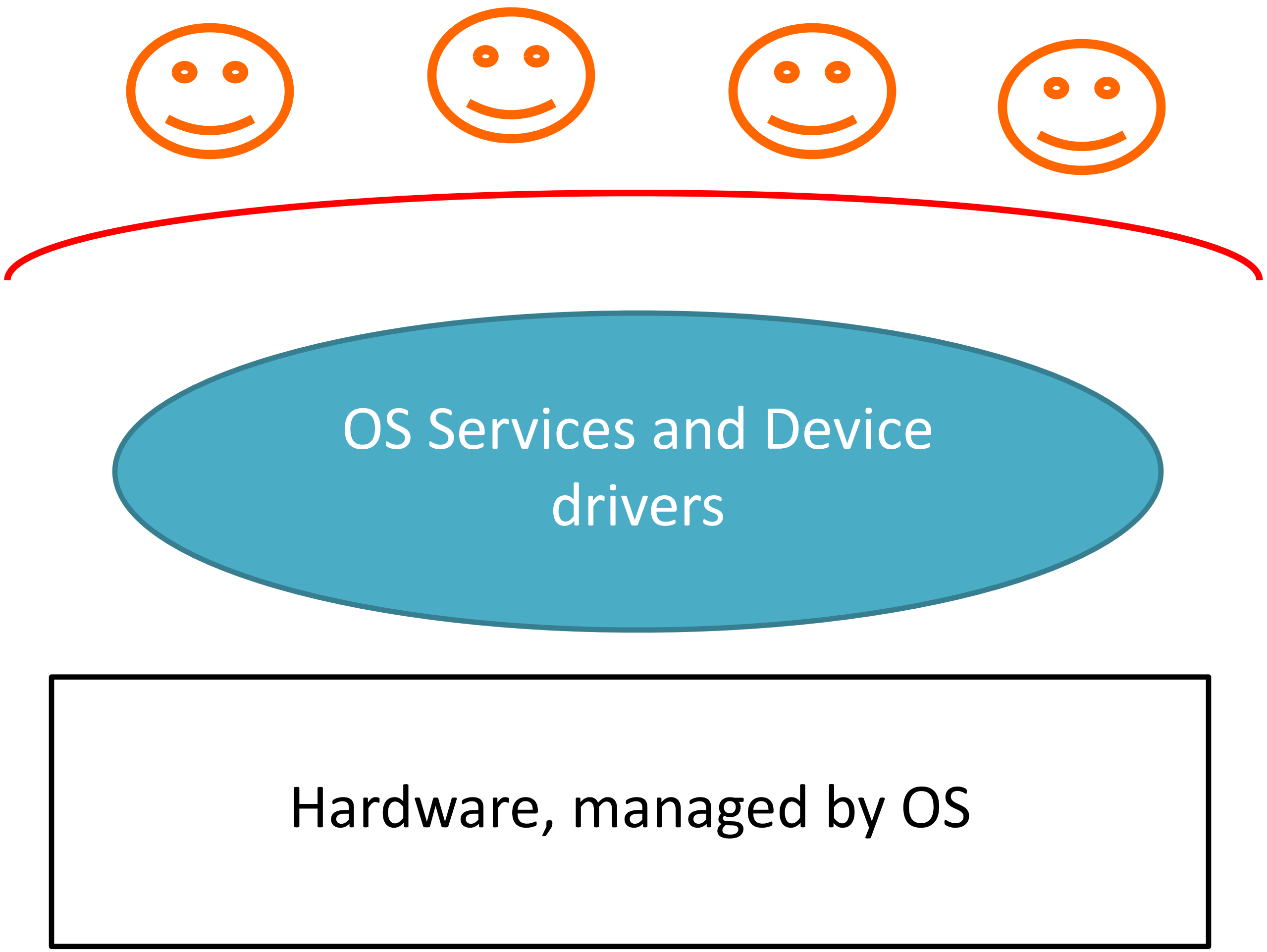
Cost of Protection

- How expensive are border crossings?
- Procedure call within the same address space:
  - Save some general-purpose registers and jump
- Mode switch:
  - Trap or syscall overhead
  - Switch to kernel stack
  - Switch some registers
  - 100s of ns
- Context switch:
  - Change address space
    - Expensive: flush TLB, caches…
  - Save and restore all registers
  - Few microsecs

OS design models

- Monolithic Kernel
- Library OS
- Micro Kernel

Monolithic Kernel



Applications

Each app has its own address space

User/kernel boundary

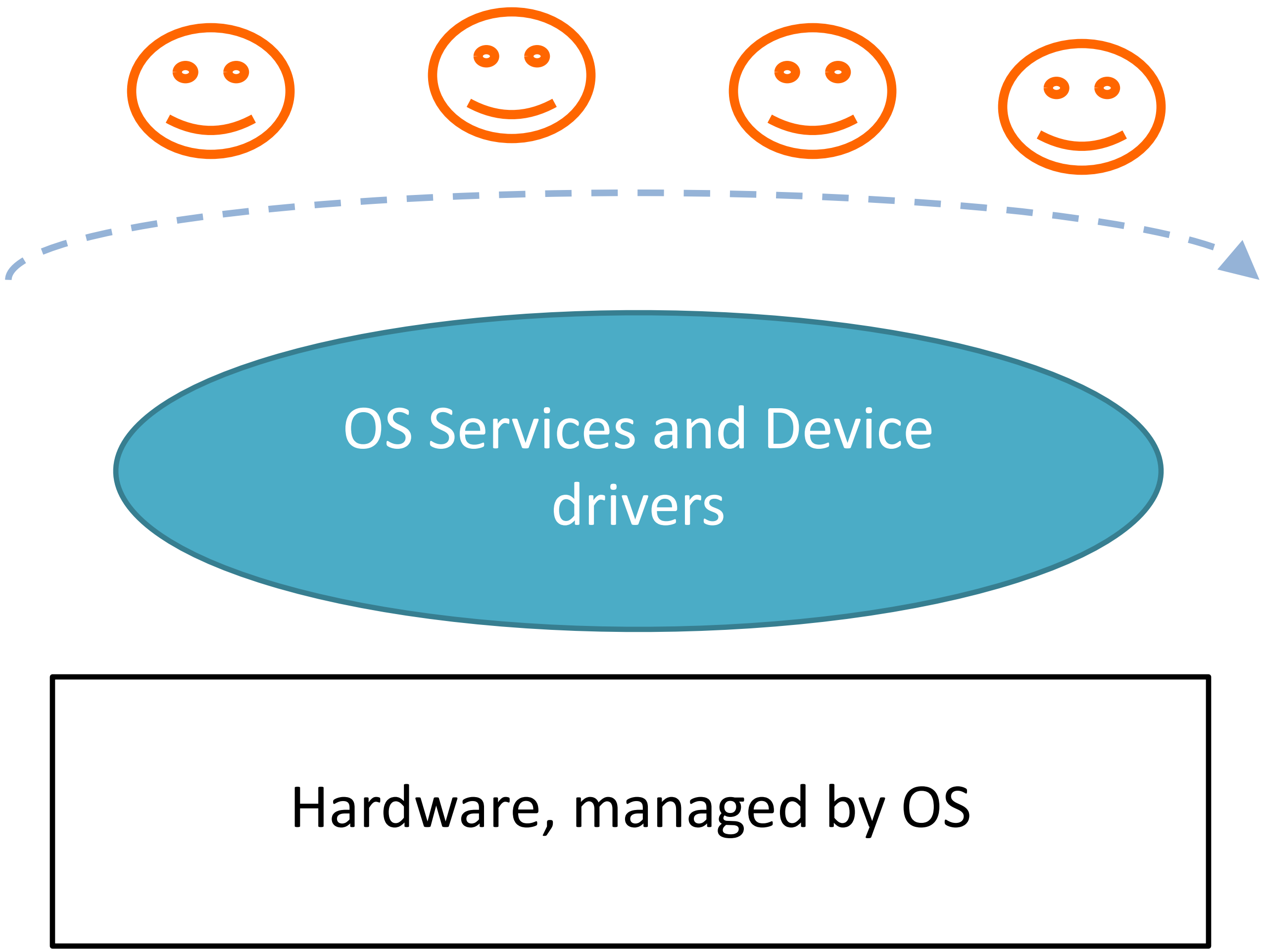
Monolithic kernel

OS in its own address space

All OS services are consolidated to reduce performance loss

Bad extensibility

Library OS (DOS-like)



Applications

Single address space. No user/kernel mode switching

Applications have direct control on hardware → Fast

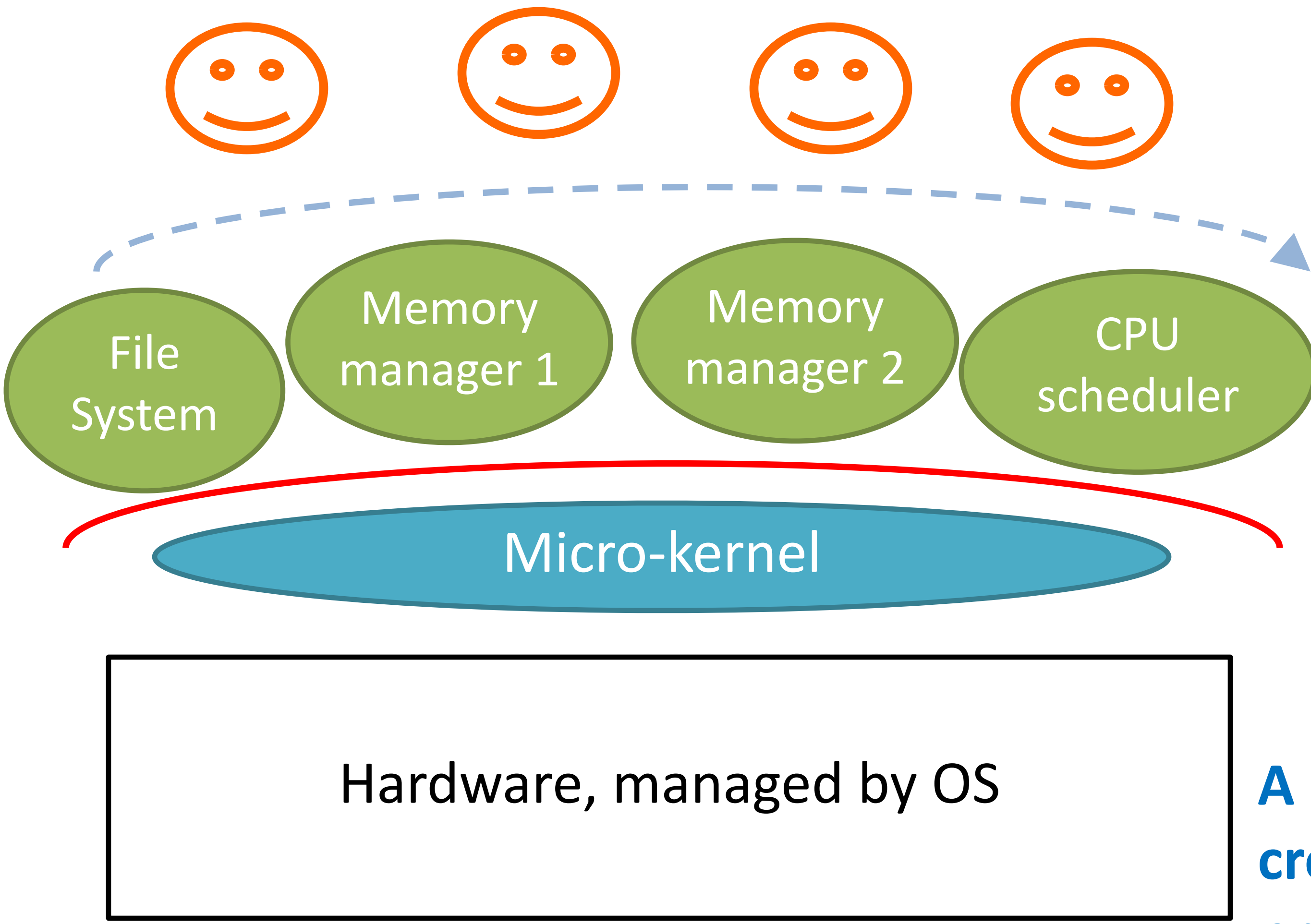
Bad protection

No protection between applications and OS

OS design models

- Dos-like library kernel
  - No protection
- Monolithic kernel
  - Reduce performance loss by consolidating all services, thus reducing the number of border crossings
- Extensibility?

Micro-kernel for extensibility



Applications

Each app has its own address space

OS Services

Each service has its own address space
Non-privileged mode!

Microkernel

Runs in privileged mode
Essential abstractions (IPC, address space)

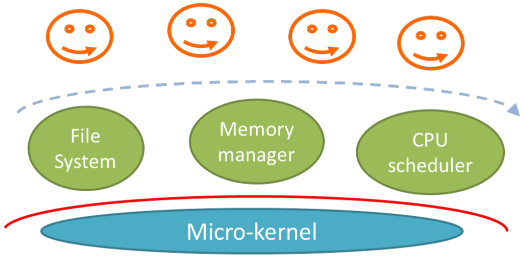
A lot of boarder crossings to ensure protection!

Micro-kernel for extensibility

- Potential performance loss
  - A lot of boarder crossings (mode & context switching)
  - Separate address spaces for OS services
    - Explicit cost: address space switching cost
    - Implicit cost: change in locality (recall caches in memory hierarchy)

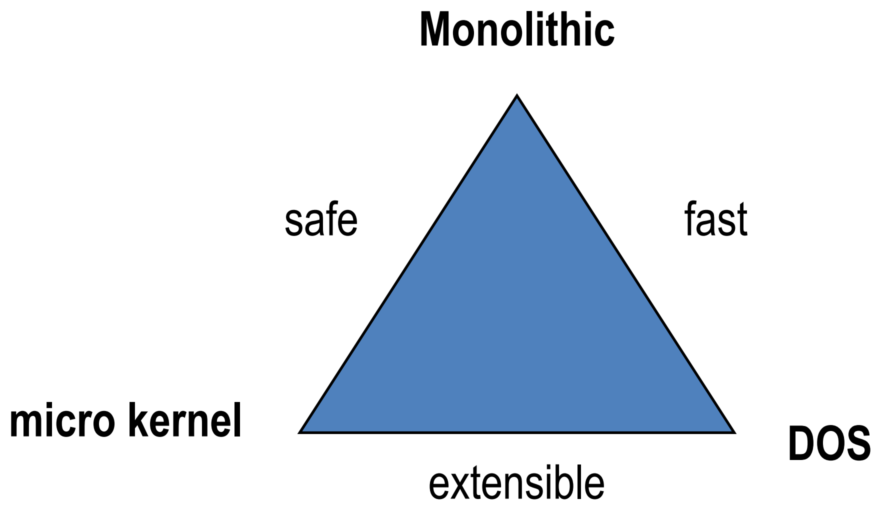
Micro-kernel for extensibility

- Potential performance loss
  - A lot of boarder crossings (mode & context switching)
  - Separate address spaces for OS services
    - Explicit cost: address space switching cost
    - Implicit cost: change in locality (recall caches in memory hierarchy)
- Let's consider an example of a file system call
  - Application uses system call to microkernel
  - Microkernel sends message to file server
  - File server does work, then uses IPC to send results back to application
  - Finally switch back to app
  - Each step is a <u>border crossing</u>



Comparison

- Library OS (DOS like):
  - Very good performance and extensibility
  - Bad (no) protection
    - Unacceptable for a general-purpose OS
- Monolithic kernels:
  - Good performance and protection
  - Bad extensibility
- Microkernels
  - Very good protection
  - Good extensibility
  - Bad performance



What should an extensible OS do?

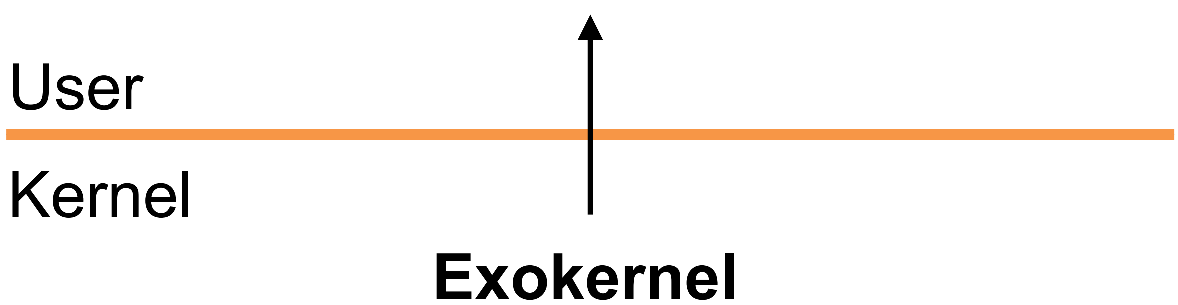
- It should be thin (like micro-kernel)
  - Only mechanisms
  - no policies; they are defined by extensions
- Fast access to resources (like DOS)
  - Eliminate border crossings
- Flexibility (like micro-kernel) without sacrificing protection or performance (like monolithic)

- Basically, *fast, protected and flexible*

Exokernel

Exokernel: An Operating System Architecture for Application-Level Resource Management

Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A
{engler, kaashoek, james}@lcs.mit.edu



- Key idea: make kernel barrier as low as possible

Context (1990s)

- Windows (Win95) was dominating the market
  - MacOS (few %)
  - Unix market (few %)
- OS research limited impact
  - "Is OS research dead?"
- A set of papers, including SPIN and Exokernel, represent an effort to reboot the OS research, particularly OS structure

Main Challenge

- Fixed interfaces/abstractions
  - Fixed interfaces provide protection yet hurt performance
  - Deny domain-specific optimization
- Exokernel:
  - *"Fixed high-level abstractions hurt application performance because there is no single way to abstract physical resources or to implement an abstraction that is best for all applications."*
- Idea: make kernel barrier as low as possible

Problems in traditional OS kernels

Problems in traditional OS kernels

- Extensibility
  - Fixed implementation (e.g., LRU for general page replacement)
  - Apps cannot dictate management
  - Abstractions overly general (e.g., page table structure)

Problems in traditional OS kernels

- Extensibility
  - Fixed implementation (e.g., LRU for general page replacement)
  - Apps cannot dictate management
  - Abstractions overly general (e.g., page table structure)
- Performance
  - Expensive mode/context switching
  - Hiding information of machine resources impact performance

Problems in traditional OS kernels

- Extensibility
  - Fixed implementation (e.g., LRU for general page replacement)
  - Apps cannot dictate management
  - Abstractions overly general (e.g., page table structure)
- Performance
  - Expensive mode/context switching
  - Hiding information of machine resources impact performance
- Protection and management offered with sacrifice in extensibility and performance

Symptoms

- Very few of innovations making into commercial OSes
  - E.g., scheduler activations, efficient IPC, new virtual memory policies, …
- Apps struggling to get better performance
  - Apps knew better how to manage and utilize resources, yet the OS was standing in the way

Exokernel Philosophy

- A nice illustration of the end-to-end argument:
  - "general-purpose implementations of abstractions force applications that do not need a given feature to pay substantial overhead costs.''
  - *"An exokernel should avoid resource management. It should only manage resources to the extent required by protection (i.e., management of allocation, revocation, and ownership)."*
  - Kernel just safely exposes resources to apps
  - Apps implement everything else, e.g., interfaces/APIs, resource allocation policies

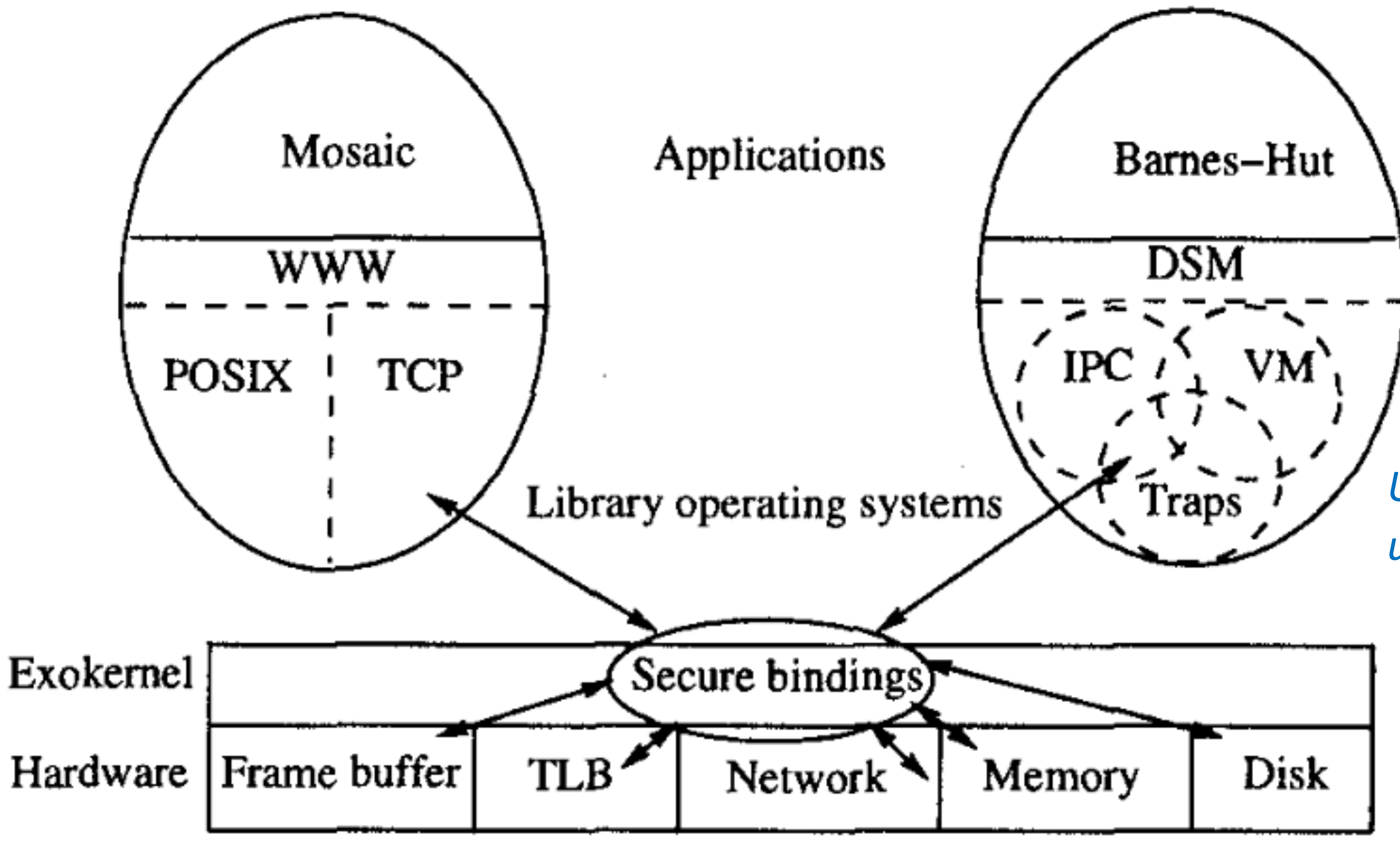
Exokernel Ideas

- Decouples the authorization to a hardware resource from its actual use
  - Kernel: resource sharing, not policies
- Higher-level abstractions are implemented in applications
  - Each application has its own Library OS
  - Exokernel grants hardware resources to Library OS
  - Library OS implements resource management policies
- Safety ensured by secure bindings
  - Safely expose machine resources

Exokernel Principles

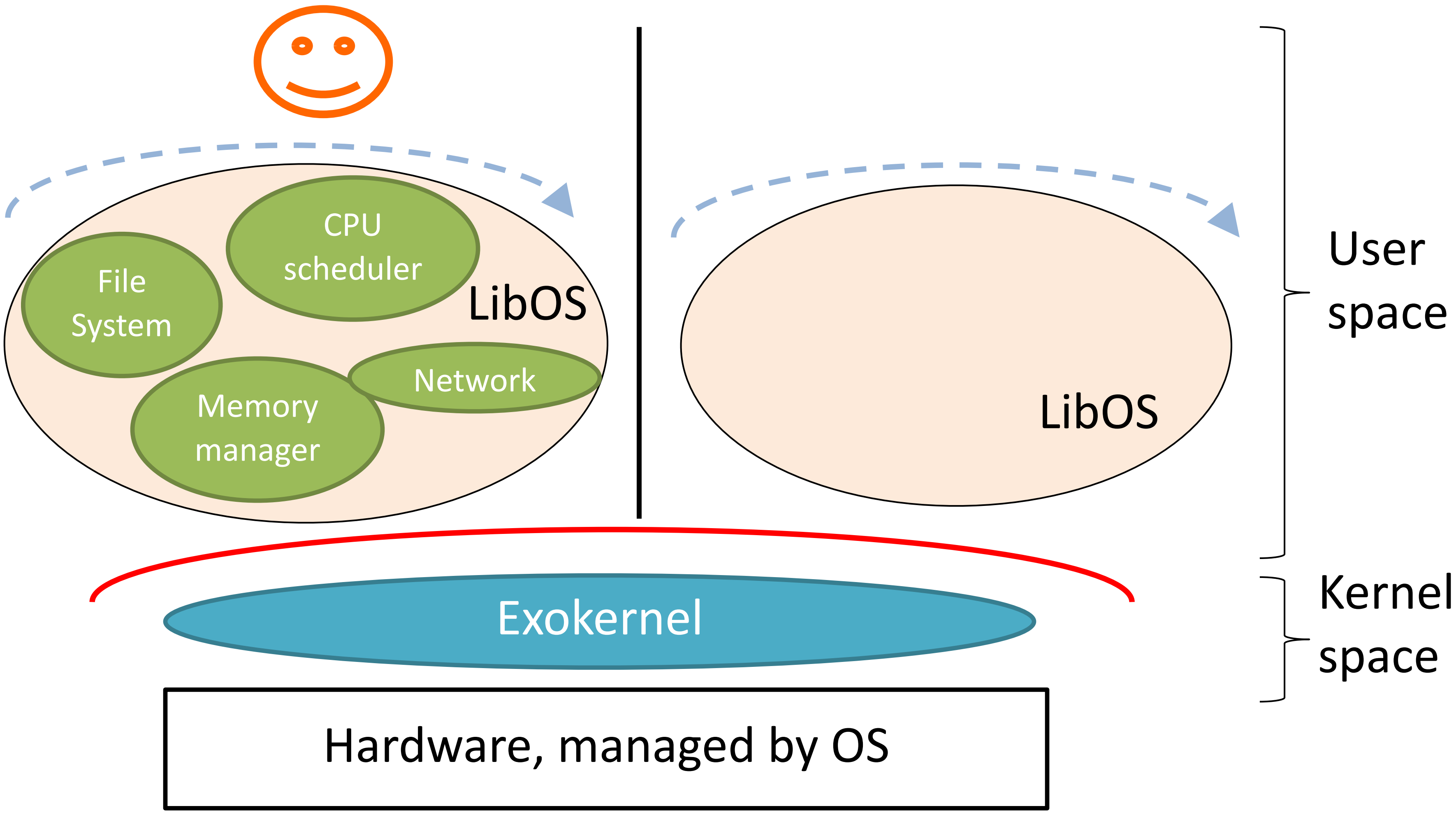
- Separate protection and management
  - Provide low-level primitives, e.g., disk blocks, context identifiers, TLB, etc.
- Expose names
  - Export physical names wherever possible, e.g., a physical page number, disk blocks, etc.
- Expose allocation
  - Apps allocate resources explicitly
- Expose revocation
  - Let apps choose which instance of a resource to give up

Exokernel Architecture



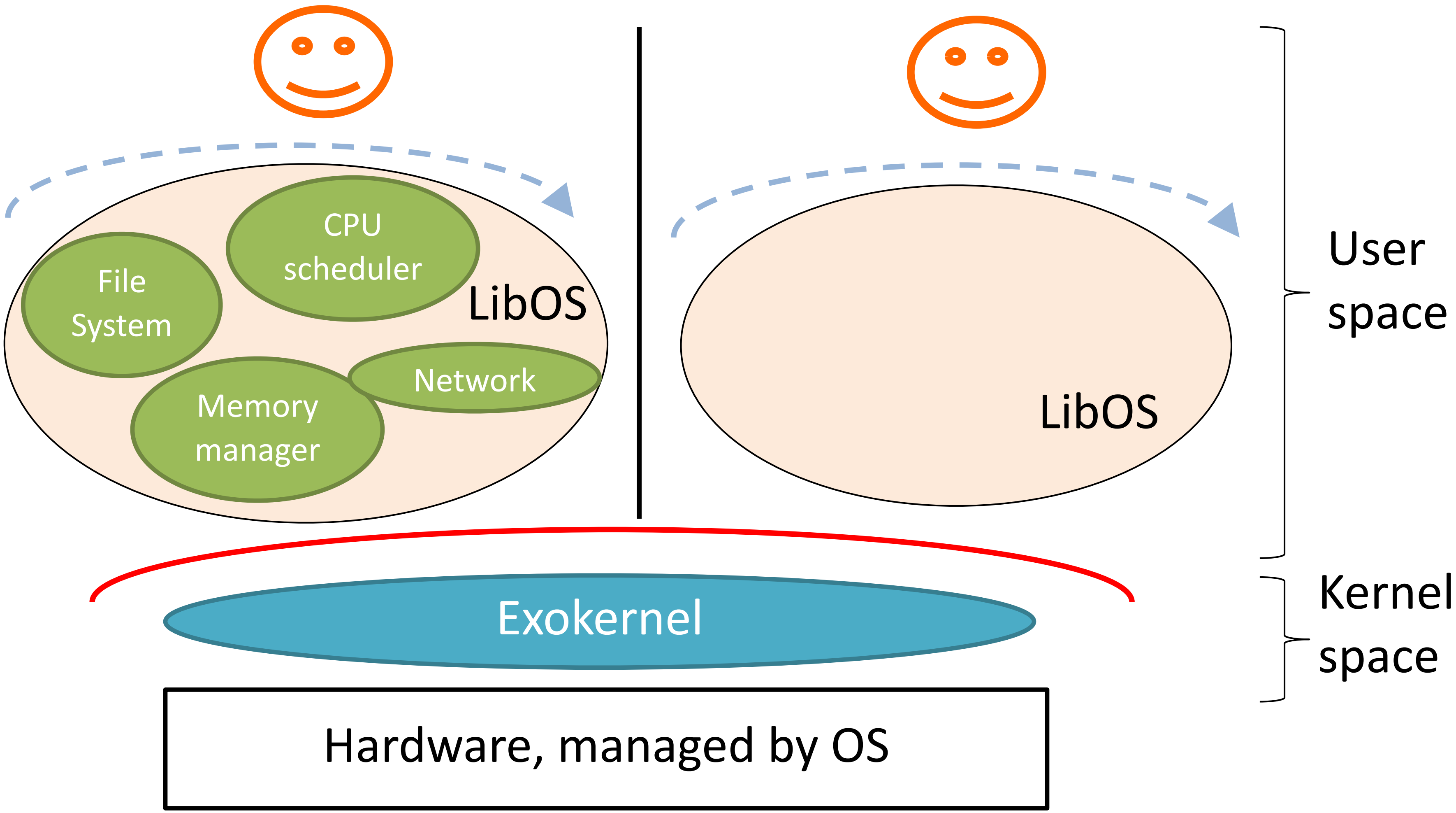
User-level, untrusted

Exokernel



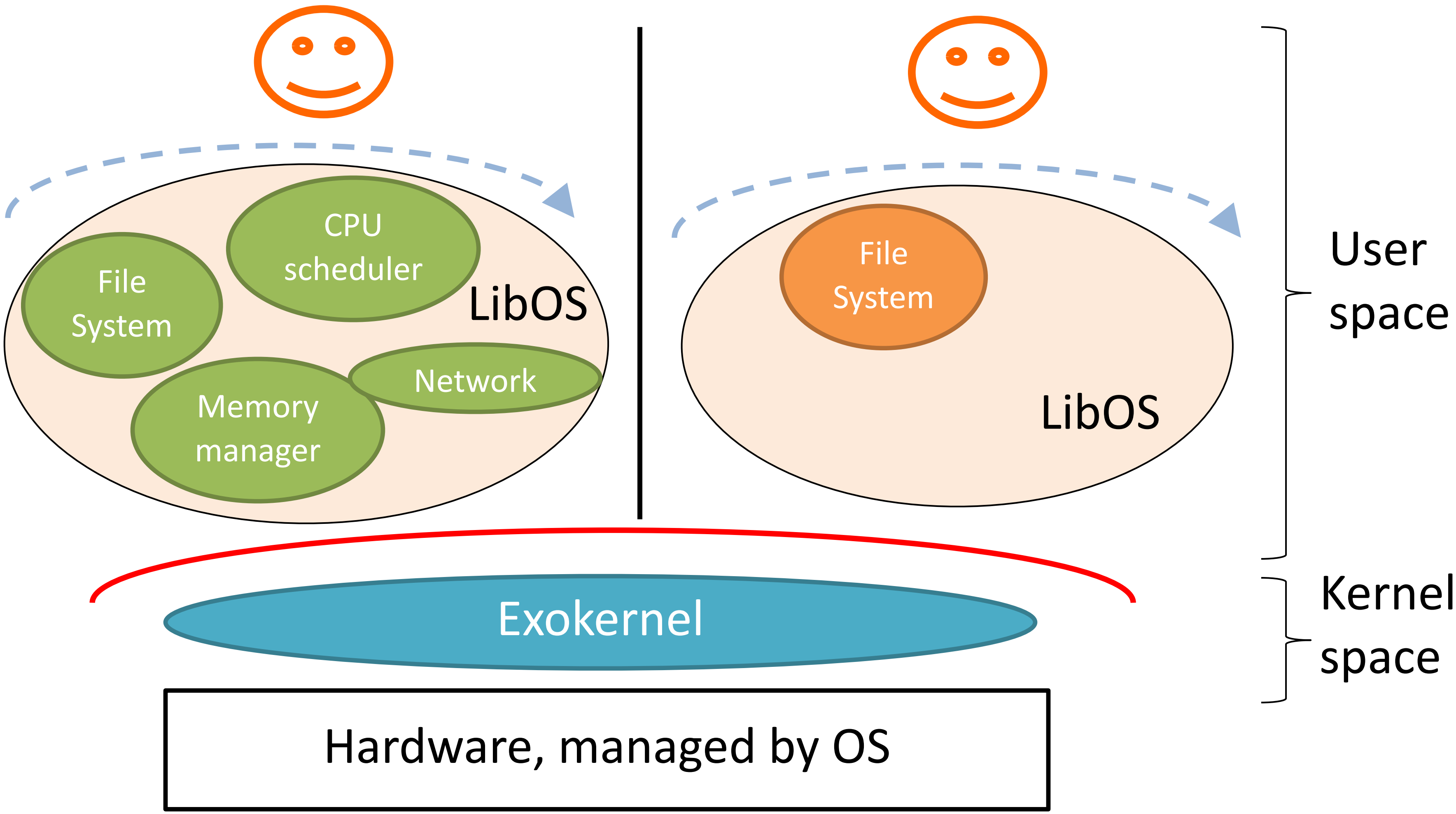
CPU
scheduler
File
System
LibOS
Network
Memory
manager
LibOS
User
space
Exokernel
Kernel
space
Hardware, managed by OS

Exokernel

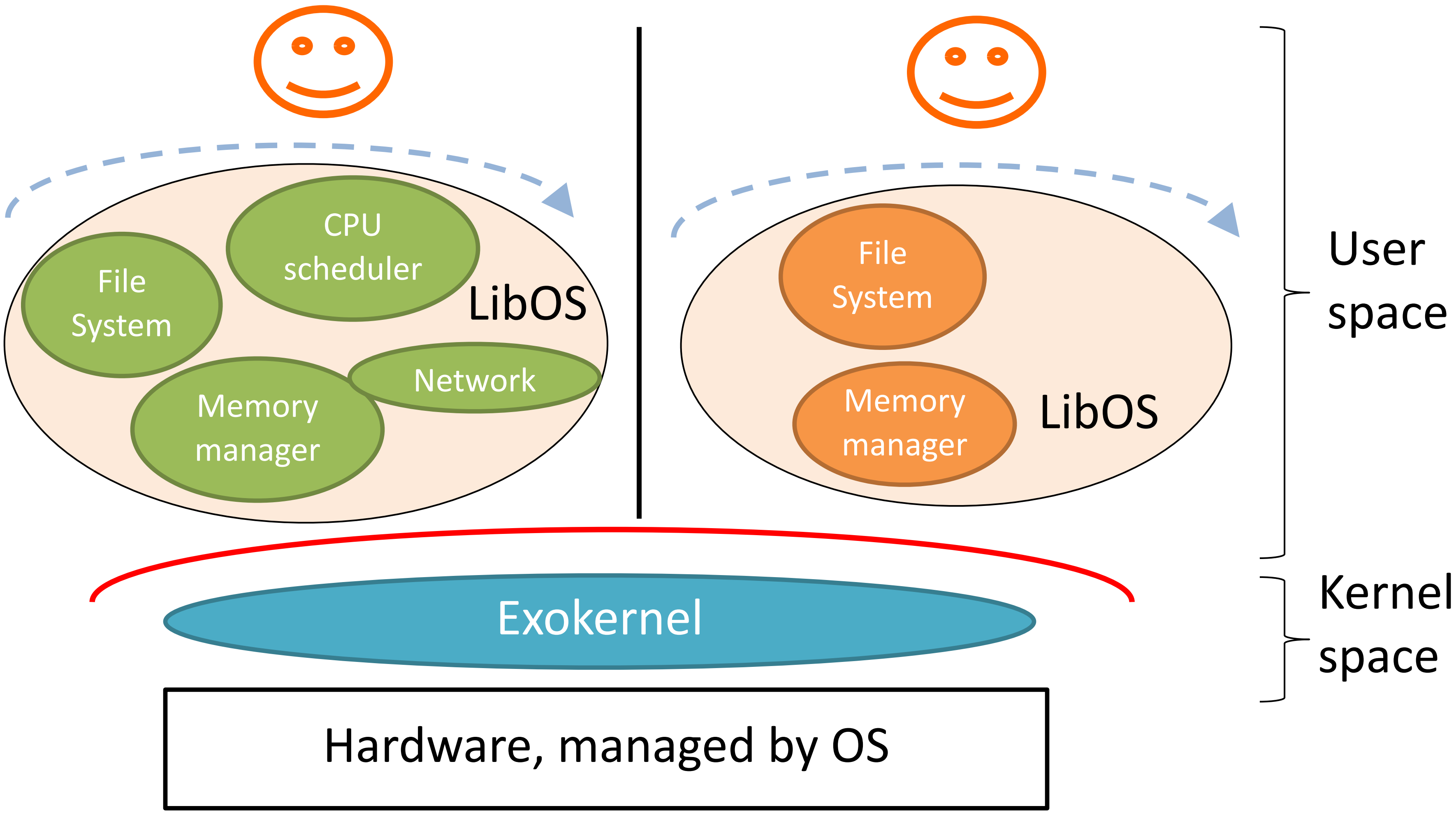


CPU scheduler
File System
LibOS
Network
Memory manager
LibOS
User space
Exokernel
Kernel space
Hardware, managed by OS

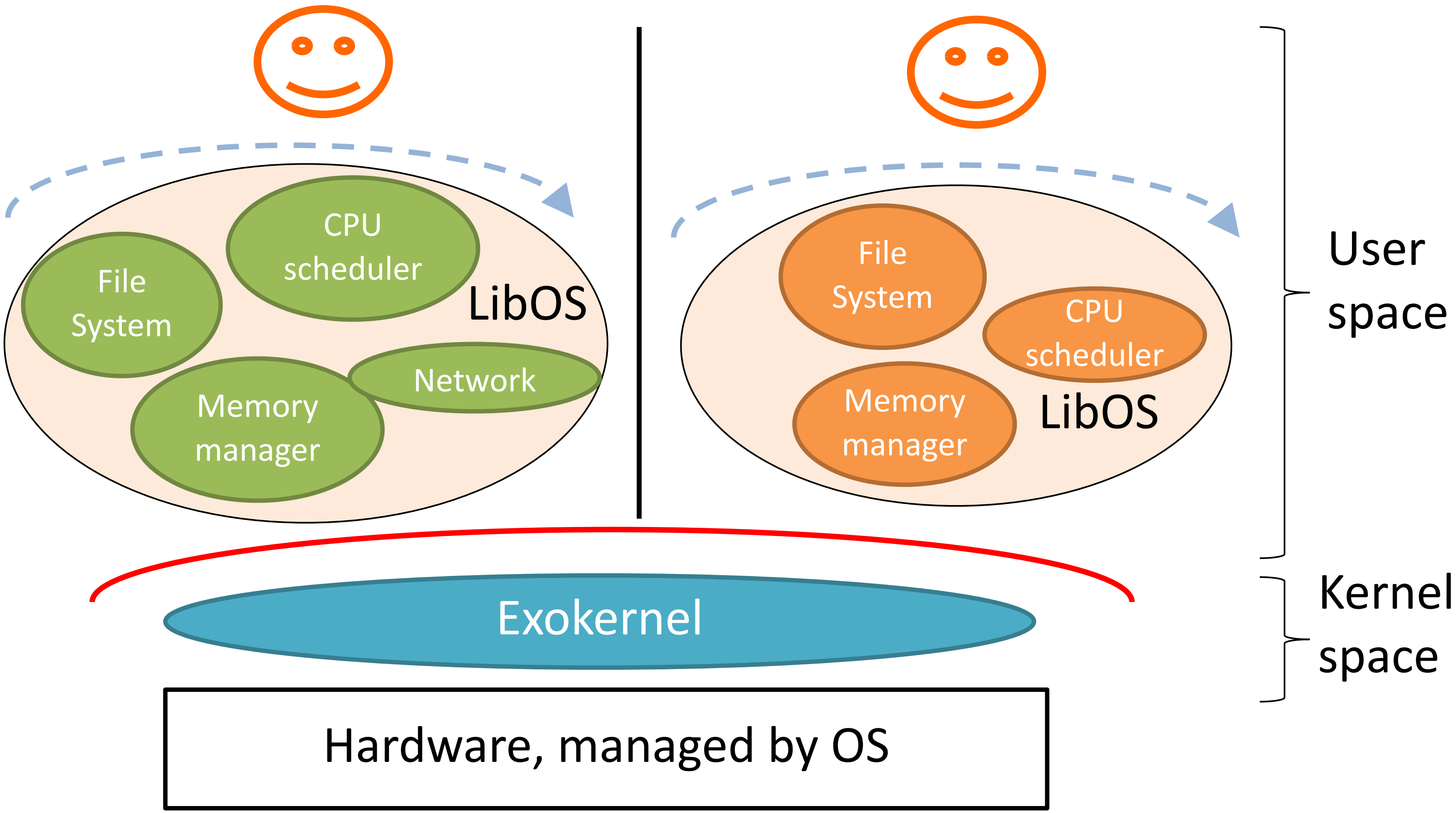
Exokernel


CPU
scheduler
File
System
LibOS
Network
Memory
manager
File
System
LibOS
User
space
Kernel
space
Exokernel
Hardware, managed by OS

Exokernel


File
System
CPU
scheduler
LibOS
Memory
manager
Network
File
System
Memory
manager
LibOS
User
space
Exokernel
Kernel
space
Hardware, managed by OS

Exokernel



File System
CPU scheduler
LibOS
Network
Memory manager
File System
CPU scheduler
Memory manager
LibOS
User space
Exokernel
Kernel space
Hardware, managed by OS

Services Provided by Exokernel

- Tracking ownership of resources
- Guarding all resource usage or binding points
- Revoking access to resources

How?

- Secure bindings
  - Allow libOSes to bind to machine resources
- Visible revocation
  - Allow libOSes to participate in resource revocation
- Abort protocol
  - Break bindings of uncooperative libOSes

Secure Bindings

- Decouples authorization from actual use of resources
- *Authorization* performed only at bind time
  - E.g., a libOS must translate a virtual to physical address
- *Protection checks* performed at access time
  - Simple operations without understanding details of application-level semantics & management policies
  - E.g., when the address translation is used by the TLB
- "Simply put, a secure binding allows the kernel to protect resources without understanding them"

Example resource

- TLB Entry
  - When a TLB fault occurs, a virtual-to-physical mapping is performed by LibOS
  - Binding presented to Exokernel
    - Each physical page: owner and read/write capabilities
    - Exokernel validates capabilities
  - Exokernel puts it in hardware TLB (why not by libOS?)
  - Applications (processes in LibOS) can then access it without Exokernel intervention

Implementing Secure Bindings

- Why?
  - Library OSes are untrusted
- How to implement?
  - Hardware mechanisms:
    - If appropriate hardware support is available; e.g., TLB entry
  - Software caching:
    - E.g., large software TLB in the kernel as a cache of frequent bindings
    - Avoid context switch when exokernel switches among libOSes
  - Downloading application code to improve performance:
    - Bindings are invoked on every event to determine ownership and kernel actions (e.g., packet filter)
    - Avoid boarder crossing

Secure Binding Example

- Multiplexing physical memory
  - A libOS allocates a physical memory page
  - Exokernel creates a secure binding for that page by recording capabilities: ownership, R/W permissions (authorization at bind time)
  - Guards every access to a physical memory page by checking capabilities (protection at access time)

Secure Bindings via Downloading Code

- libOSes "download" code into exokernel
  - Providing code for exokernel to execute on the apps behalf
  - E.g., code to determine which pages to swap out when memory is needed
  - E.g., code to schedule multiple threads within the process

Secure Bindings via Downloading Code

- libOSes “download” code into exokernel
  - Providing code for exokernel to execute on the apps behalf
  - E.g., code to determine which pages to swap out when memory is needed
  - E.g., code to schedule multiple threads within the process
- Performance boost by eliminating boarder crossings
- Runtime of certain operations predictable a priori

Secure Bindings via Downloading Code

- libOSes "download" code into exokernel
  - Providing code for exokernel to execute on the apps behalf
  - E.g., code to determine which pages to swap out when memory is needed
  - E.g., code to schedule multiple threads within the process
- Performance boost by eliminating boarder crossings
- Runtime of certain operations predictable a priori
- Cons: potentially unsafe!!
  - Lots of other work around this time on extensible OS designs that tried to implement this feature (with various degree of success)

Visible Resource Revocation

- Traditional OS: resources revoked invisibly
- Exokernel: visible deallocation of resource
  - So that library OS has a chance to react
    - e.g. under memory pressure, libOS can choose a victim page
  - But could be less efficient when revocations happen frequently
    - Can be combined with invisible revocation (esp. when it happens frequently): e.g., process address space identifiers which is a stateless resource

Abort Protocol

- Visible resource revocation is good... But what if application does not cooperate?
  - "An exokernel must also be able to take resources from libOSes that fail to respond satisfactorily to revocation requests"
- Abort protocol
  - Forced resource revocation
  - Break secure bindings to the resource
  - Uses 'repossession vector' to record the forced loss of resources

Managing core services

- Virtual memory
  - Secure binding: using self-authenticating capabilities
  - When accessing page, owner needs to present capability
  - Page owner can change capabilities associated and deallocate it

Managing core services: scheduling

- Processor time represented as linear vector of time slices
  - Round robin allocation of slices
- Secure binding: allocate slices to LibOS
  - Simple, powerful technique: donate time slice to a particular process
  - A LibOS can donate unused time slices to its process of choice
- If process takes excessive time, it is killed (revocation)

Evaluation

- A full implementation; it works and scales
- How to make sense from the quantitative results?
  - Absolute numbers are typically meaningless given that we are part of a bigger system
    - Trends are what matter
- Emphasis is on space and time
  - Takeaway→ at least as good as a monolithic kernel, sometimes much faster

Conclusions

- Performance vs Extensibility vs Protection
  - DOS provided no protection. Without protective checks, you get performance and apps could directly hack the core to get extensibility.
  - Monolithic kernels implemented performance and protection, but were hard to extend
  - Microkernels provided good protection and were extensible, but performance suffered

Conclusions

- Simplicity and limited exokernel primitives can be implemented efficiently
- Hardware multiplexing can be fast and efficient
- Traditional abstractions can be implemented at application level
- Applications can create special purpose implementations by modifying libraries

Why people don't use exokernels today?

- Today's OSes give applications far greater control over low-level mechanisms than in 1990s
- LibOS concepts are used reasonably frequently (look up the Unikernel)
- Hypervisors have exokernel-like low-level interfaces to guest OSes

Discussions

- Much of computer system research is about tradeoffs
  - Efficiency v.s. Extensibility
  - Efficiency v.s. Security
  - Efficiency v.s. Fairness
  - Efficiency v.s. Correctness