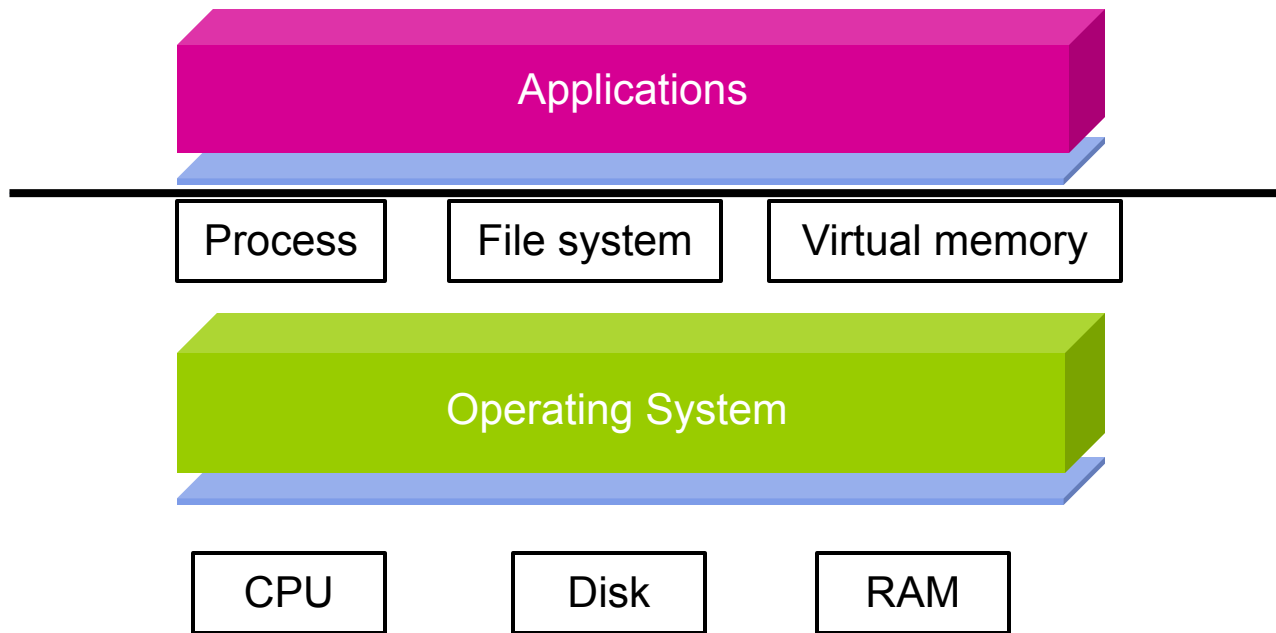


Extensible OS Design

CS 202: Advanced Operating Systems

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?

↳ User/Kernel Space
↳ Interprocess Communication


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

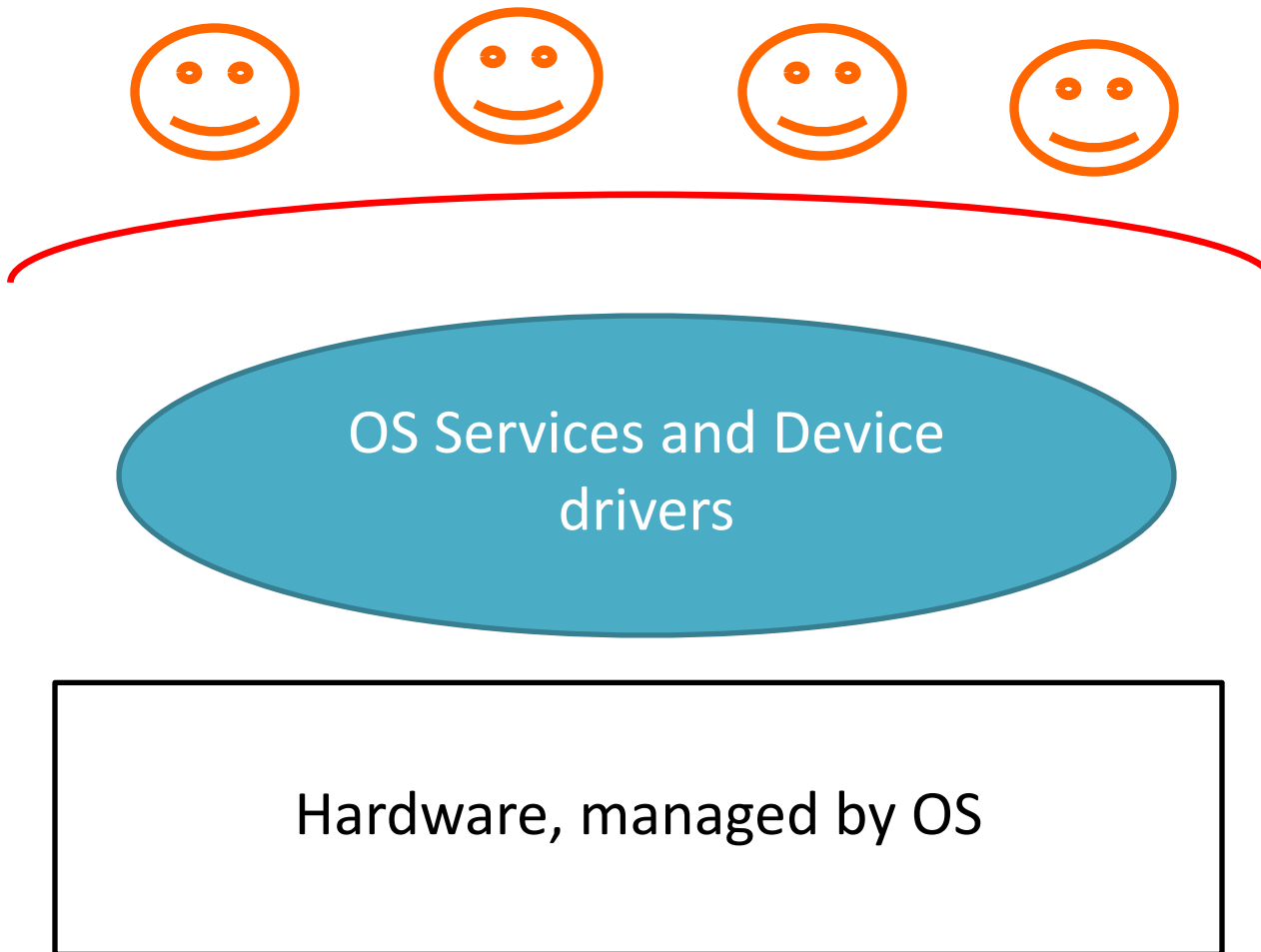
Cost of Protection

- 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  *Relatively fast*
- 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

Good for protection

User/kernel boundary

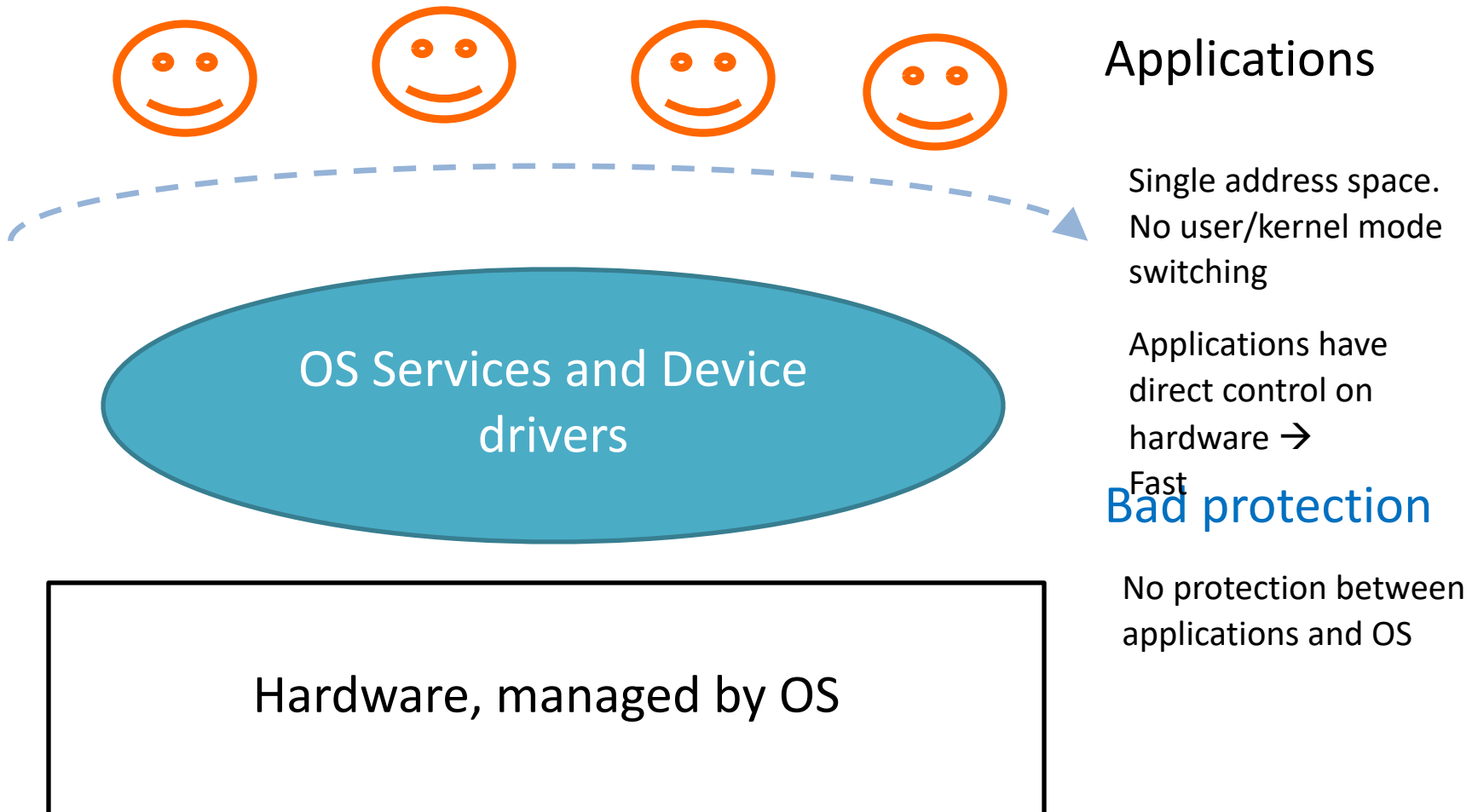
Monolithic kernel

OS in its own address space

All OS services are consolidated to reduce performance loss

Bad extensibility

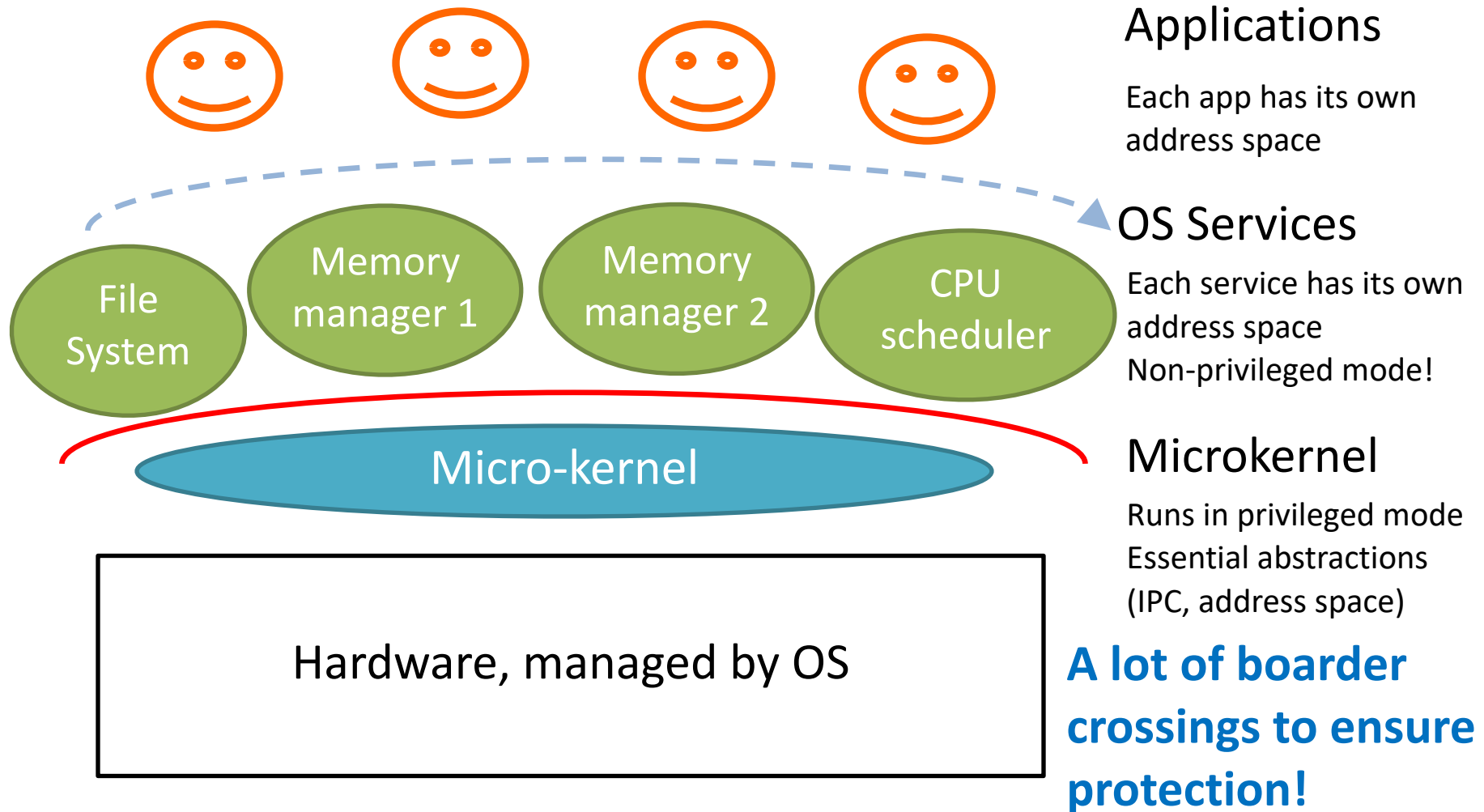
Library OS (DOS-like)



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

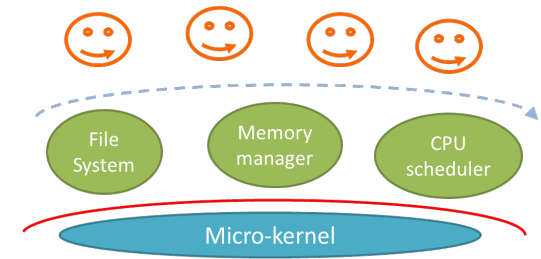


Micro-kernel for extensibility

- 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)

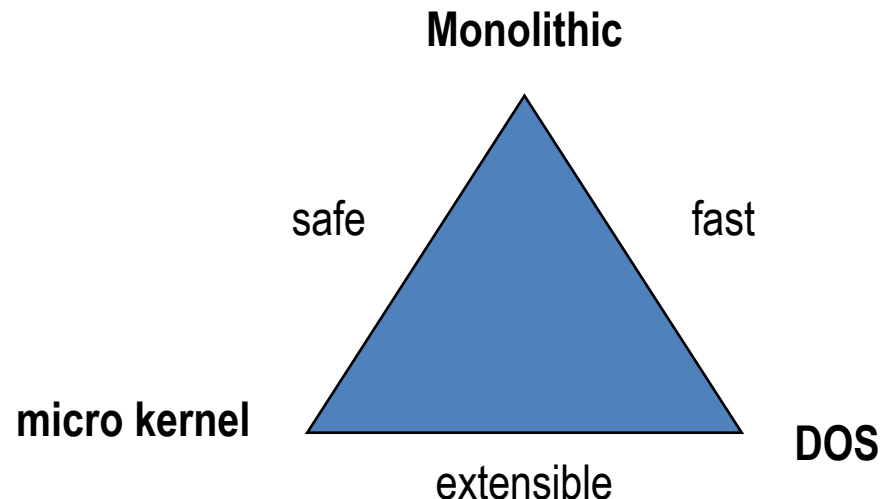
Micro-kernel for extensibility

- 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 border crossing



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*

Exposed Kernel

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`

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

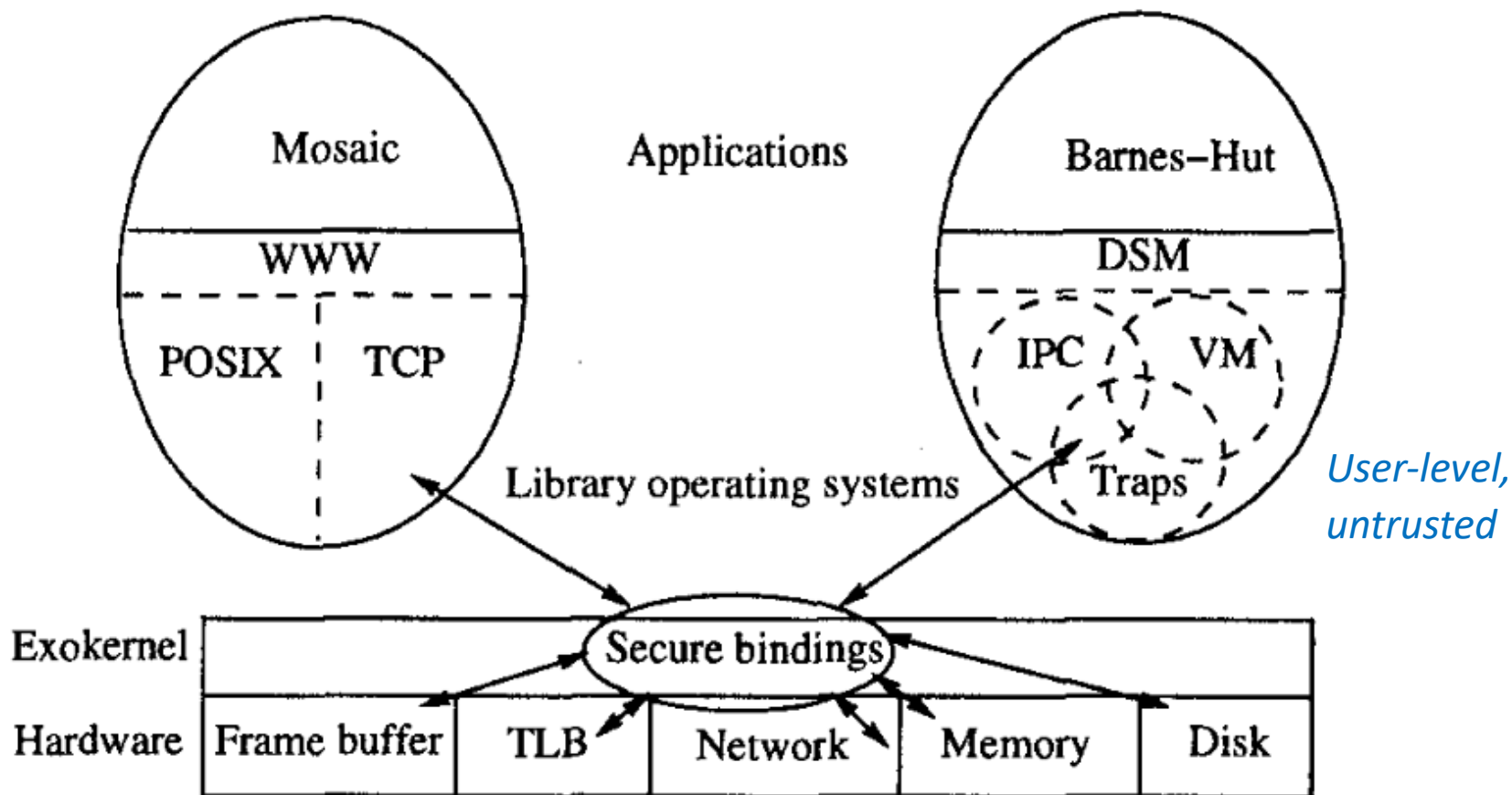
Exokernel Ideas

- 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

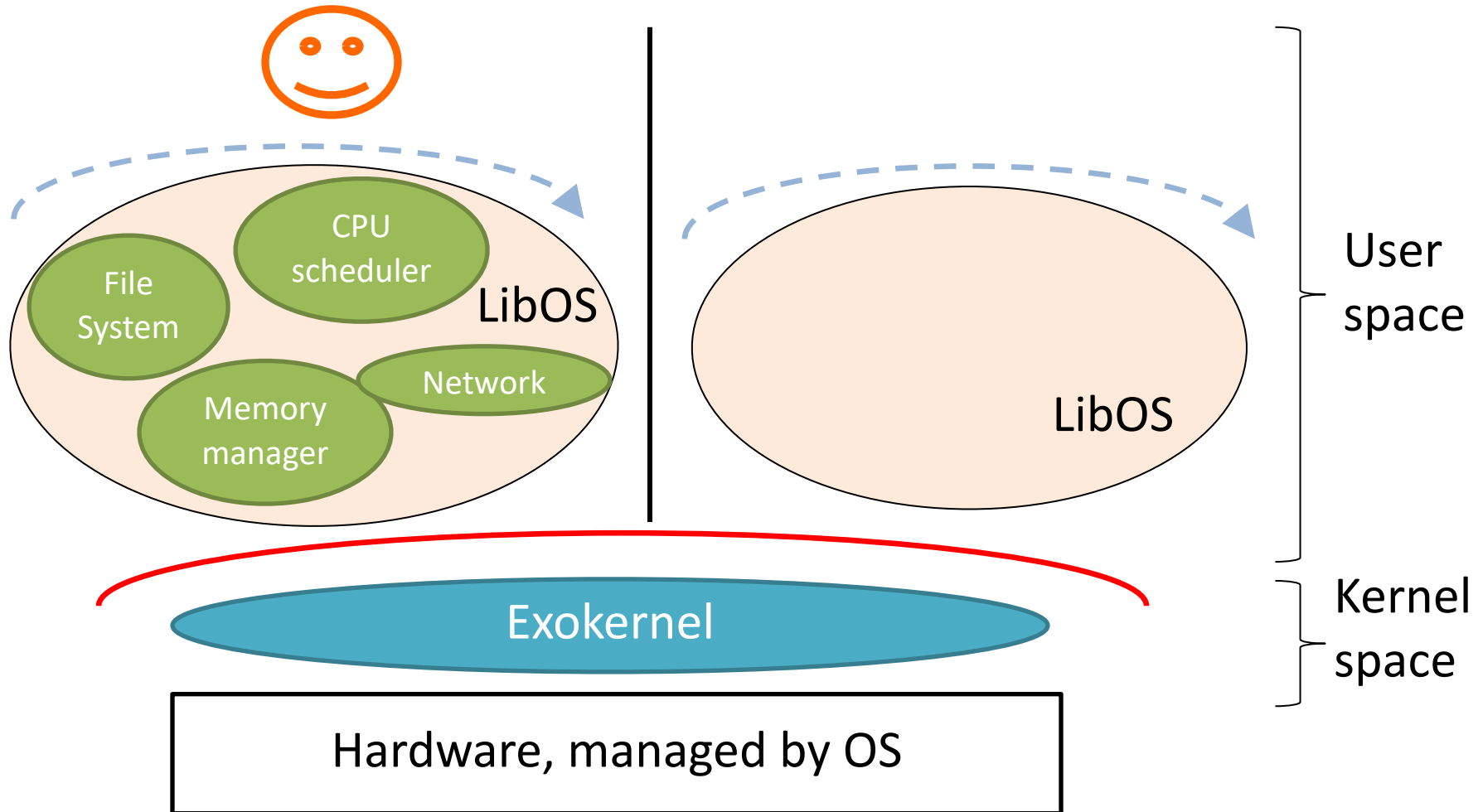
Exokernel Principles

- 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

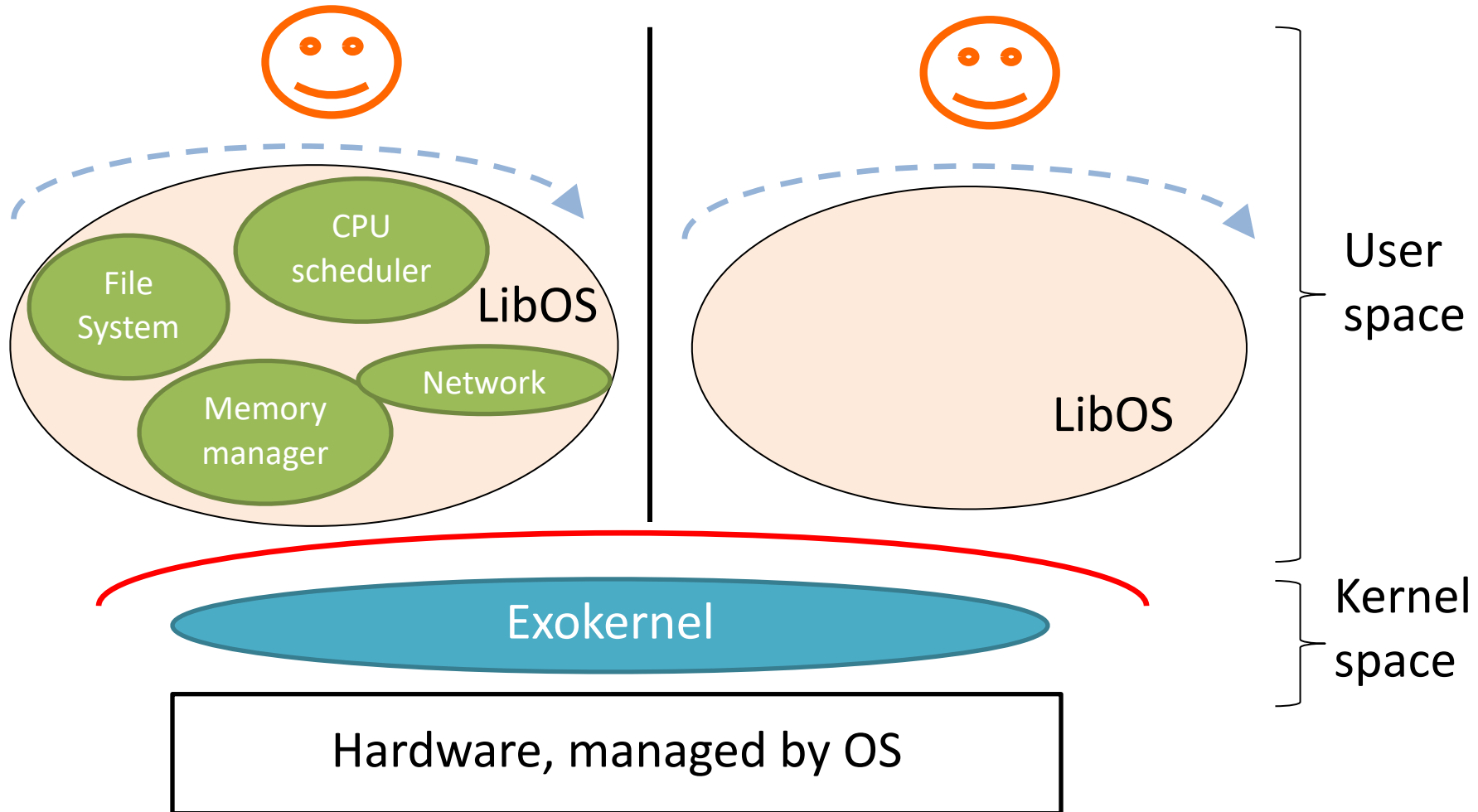
Exokernel Architecture



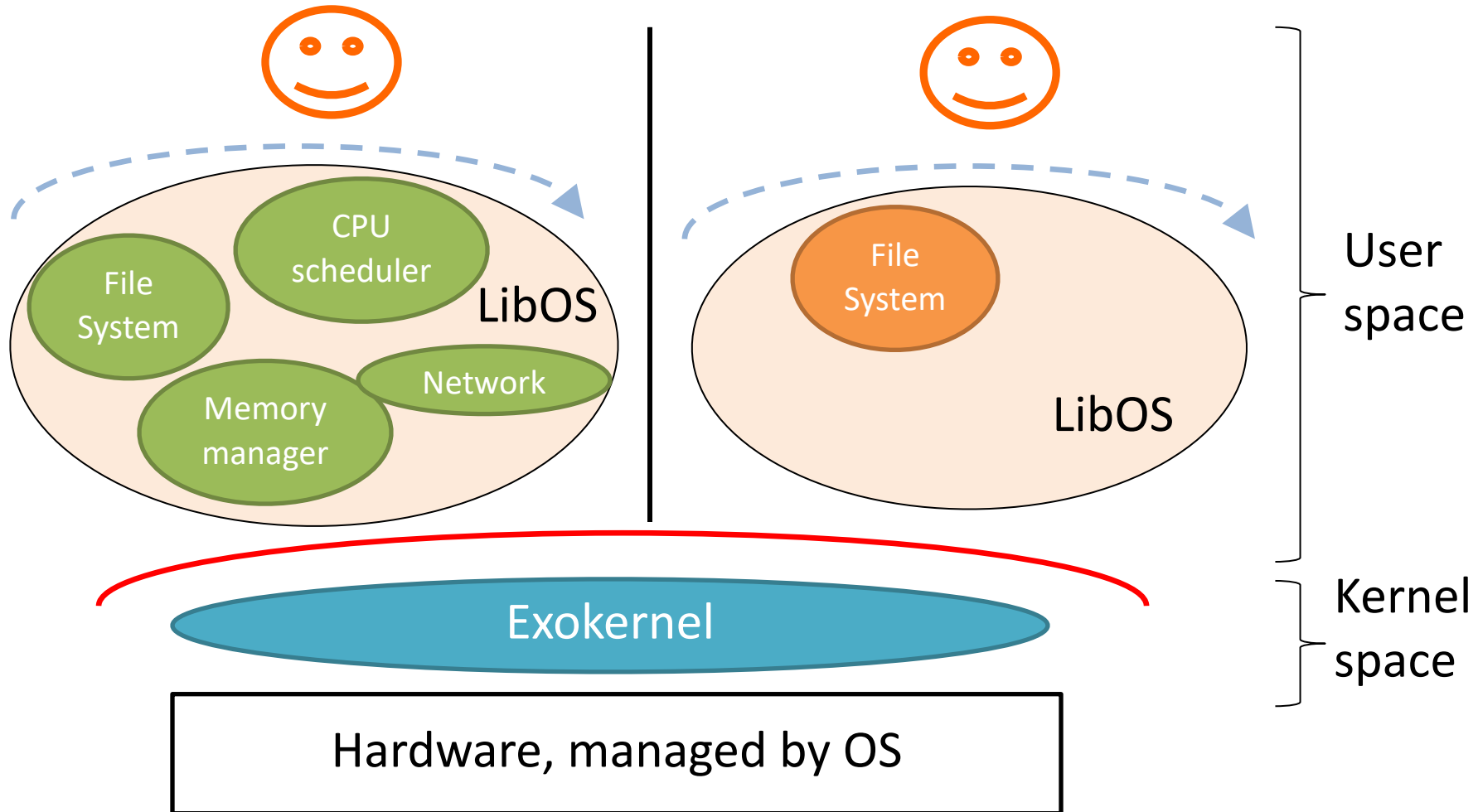
Exokernel



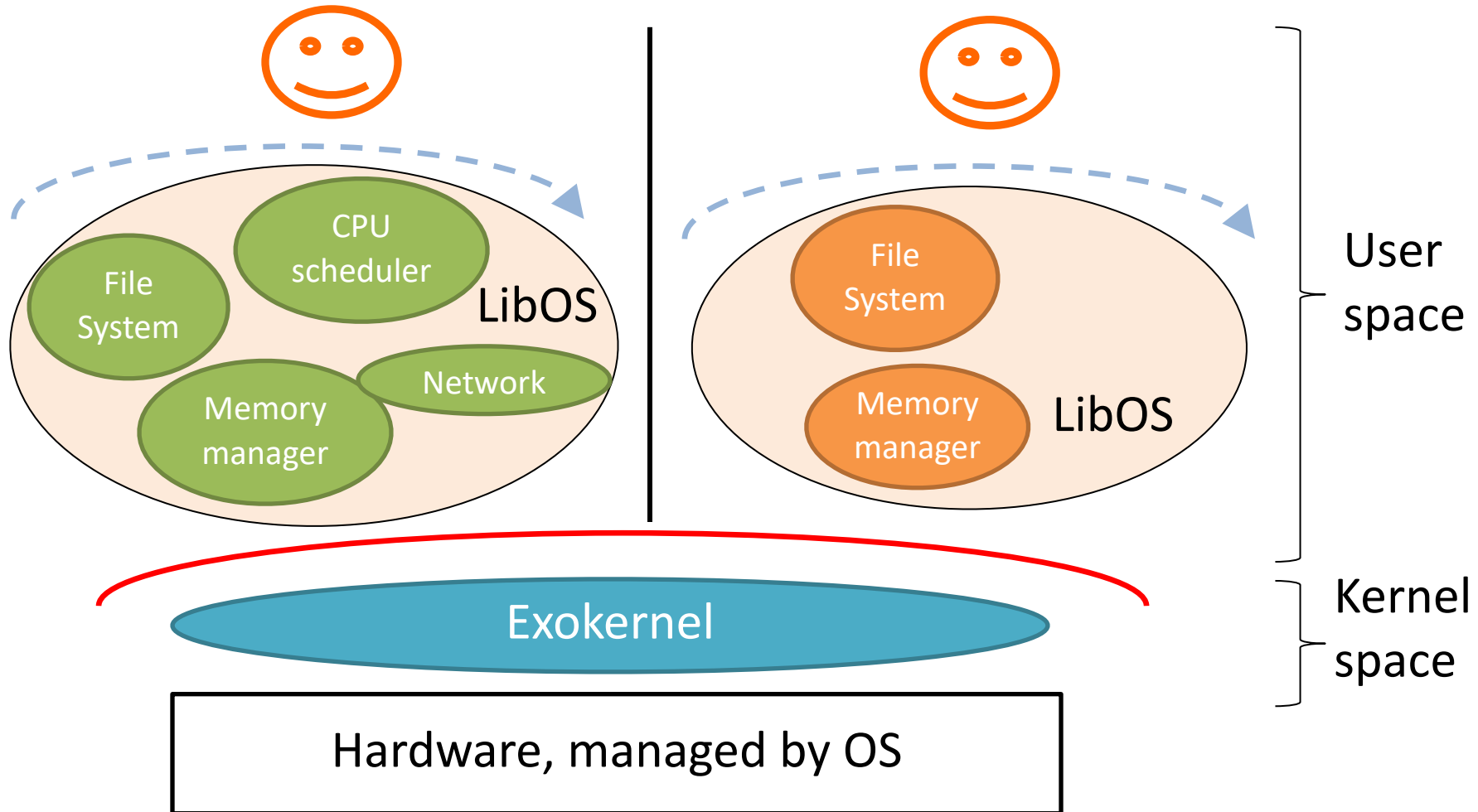
Exokernel



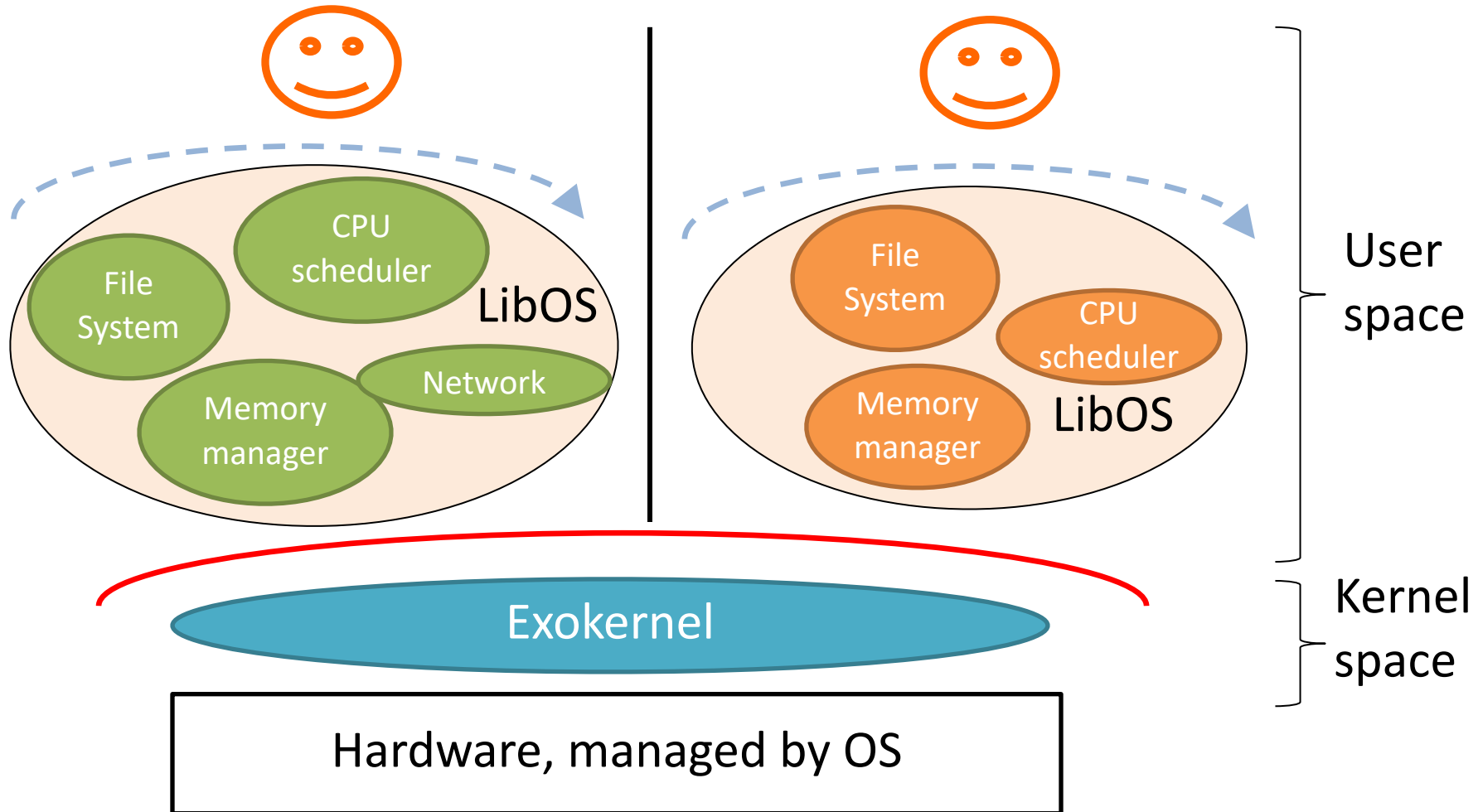
Exokernel



Exokernel



Exokernel



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

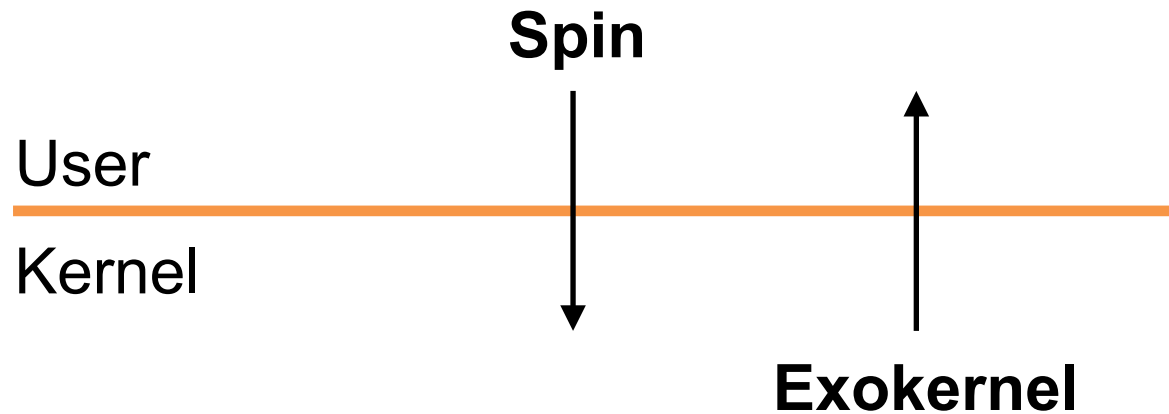
Managing core services: scheduling

- 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

The two (very different) approaches



- Spin: move application-specific functionalities into kernel
- Exokernel: make kernel barrier as low as possible

SPIN

Extensibility, Safety and Performance in the
SPIN Operating System

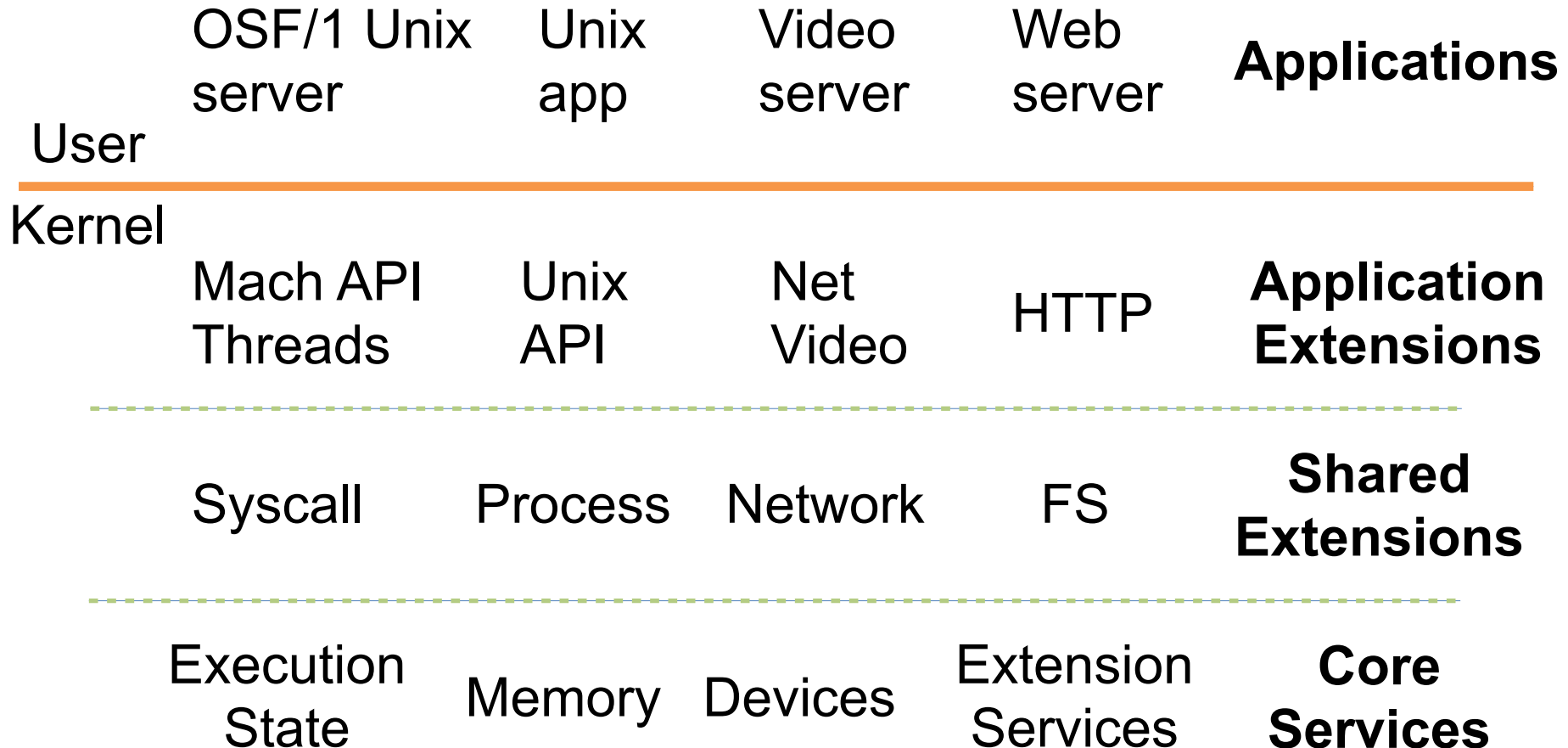
Brian N. Bershad	Stefan Savage	Przemysław Pardyak	Emin Gün Sirer
Marc E. Fiuczynski	David Becker	Craig Chambers	Susan Eggers

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Extensibility: SPIN's idea

- Co-location of kernel and extensions
 - Avoid broader crossings
 - Mechanisms in OS, policies through the extensions (e.g., Timer is a mechanism but setting timer for a user is Policy)
 - Protection??
- Language/compiler forced protection
 - Strongly typed language (Modula-3)
 - Predefined data types
 - Certain operations allowed only with specific data types
 - No cheat using pointers (e.g., no type casting of pointers in C)
 - Protection by compiler and runtime
- Logical protection domains:
 - Not relying on hardware address space!
- Generic interfaces + dynamic call binding for extensions
 - Fine-grained extensibility

Enhance performance: make extensions as cheap as procedure calls



Protection domain

- Conventional OS: use **virtual address spaces**
 - Example: a process can only access memory within a particular range of virtual addresses
 - Violation can be detected by hardware (e.g., page fault)
- But...
 - Inadequate for fine-grained protection and management (why?)
 - Large page size
 - Limited and inflexible page fault mechanism
 - Inefficient management
 - Expensive to create and slow to access

SPIN's Logical protection domains

- No longer rely on VM to enforce protection
 - A domain defines a set of names, or program symbols, that can be referenced by code with access to that domain
- **Modula-3** safety and encapsulation mechanisms
 - Type safety, automatic storage management
 - Objects, threads, exceptions and generic interfaces
- Fine-grained protection of objects using **capabilities**
 - Objects can be:
 - Hardware resources (e.g., page frames)
 - Interfaces (e.g., page allocation module)
 - Collection of interfaces (e.g., full VM)
 - Capabilities are language-supported type-safe pointers

Protection Model

→ "capability to access"

- All kernel resources are referenced by *capabilities*
 - “A *capability* is an unforgeable reference to a resource which can be a system object, an interface, or a collection of interfaces.”
 - Extensions can reference only the resources to which they have been given access
- SPIN implements capabilities directly through the use of pointers
 - Memory reference supported by the language
 - Compiler prevents pointers to be dereferenced in a way inconsistent with its type at *compile time* (Modula-3)
 - Efficiency: no run time overhead for using a pointer

Logical protection domains -- Mechanisms

- **SPIN protection domain** defines a set of names or program **symbols** that can be referenced by other domains
 - Create:
 - Initialize with object file contents and export names
 - Resolve:
 - Names are resolved between a source and a target domain
 - Once resolved, access is at memory speeds
 - Combine
 - To create an aggregate domain
- This is the key to spin – protection, extensibility and performance

```

INTERFACE Domain;

TYPE T <: REFANY;  (* Domain.T is opaque *)

PROCEDURE Create(coff:CoffFile.T):T;
(* Returns a domain created from the specified object
   file (''coff'' is a standard object file format). *)

PROCEDURE CreateFromModule():T;
(* Create a domain containing interfaces defined by the
   calling module. This function allows modules to
   name and export themselves at runtime. *)

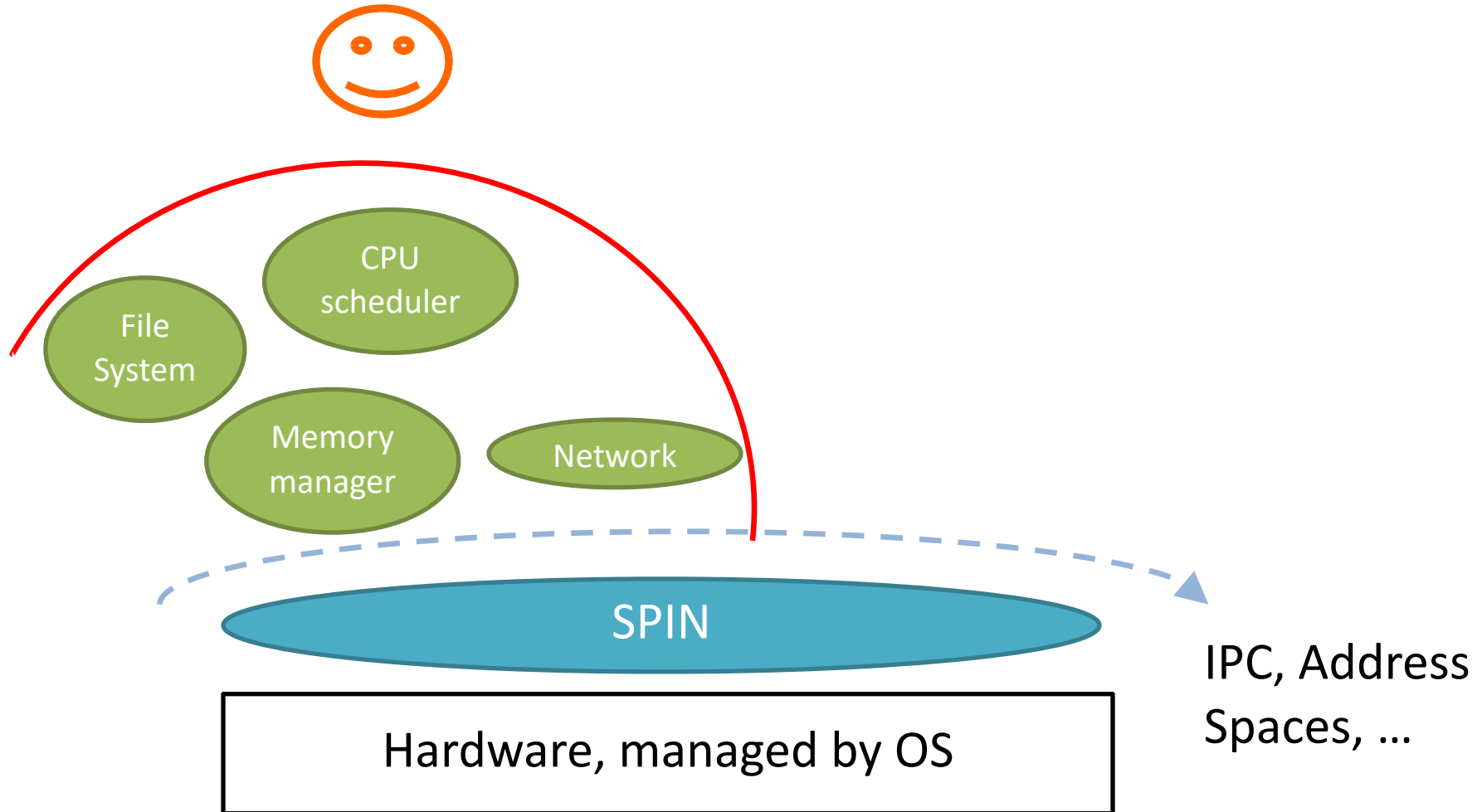
PROCEDURE Resolve(source,target: T);
(* Resolve any undefined symbols in the target domain
   against any exported symbols from the source. *)

PROCEDURE Combine(d1, d2: T):T;
(* Create a new aggregate domain that exports the
   interfaces of the given domains. *)

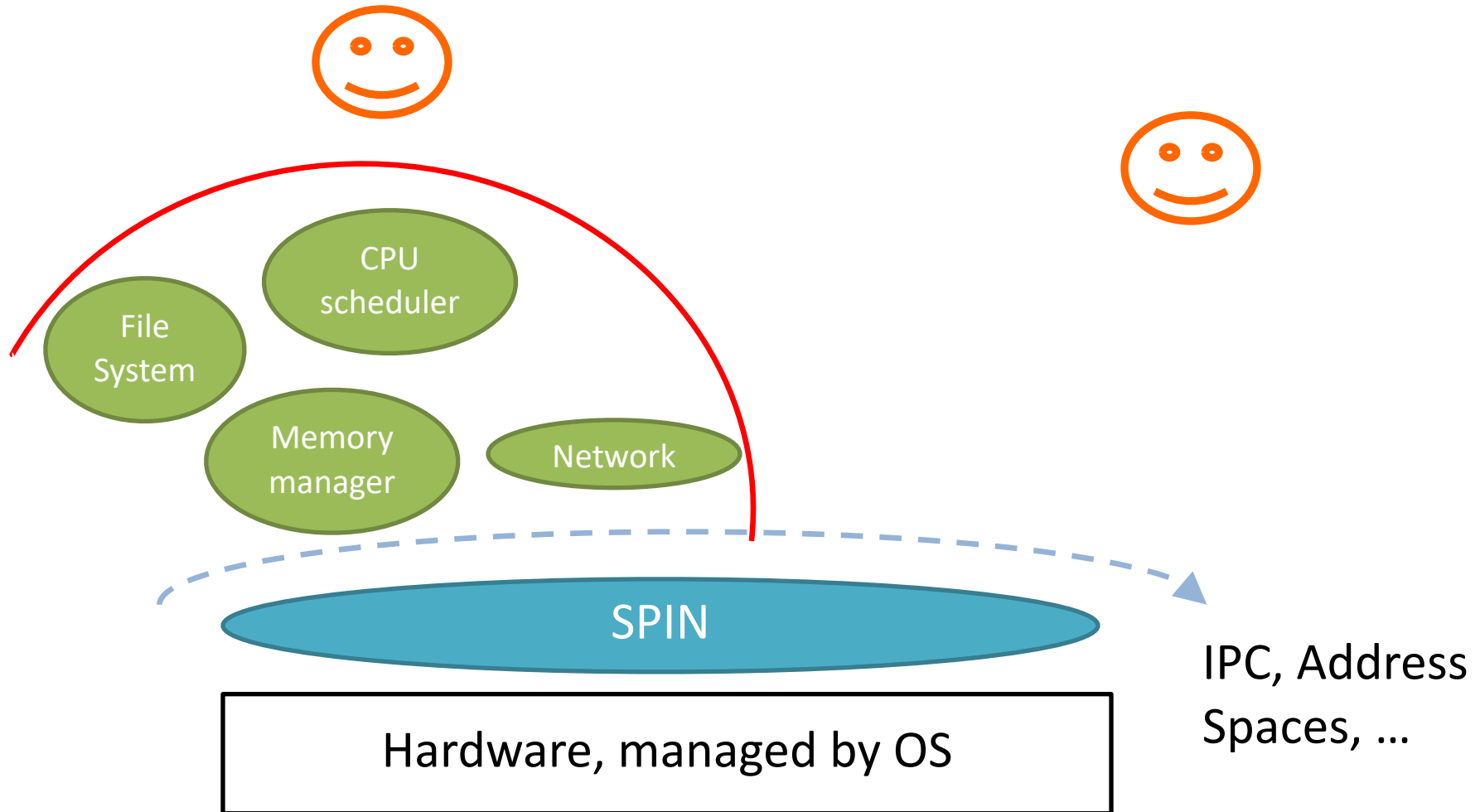
END Domain.

```

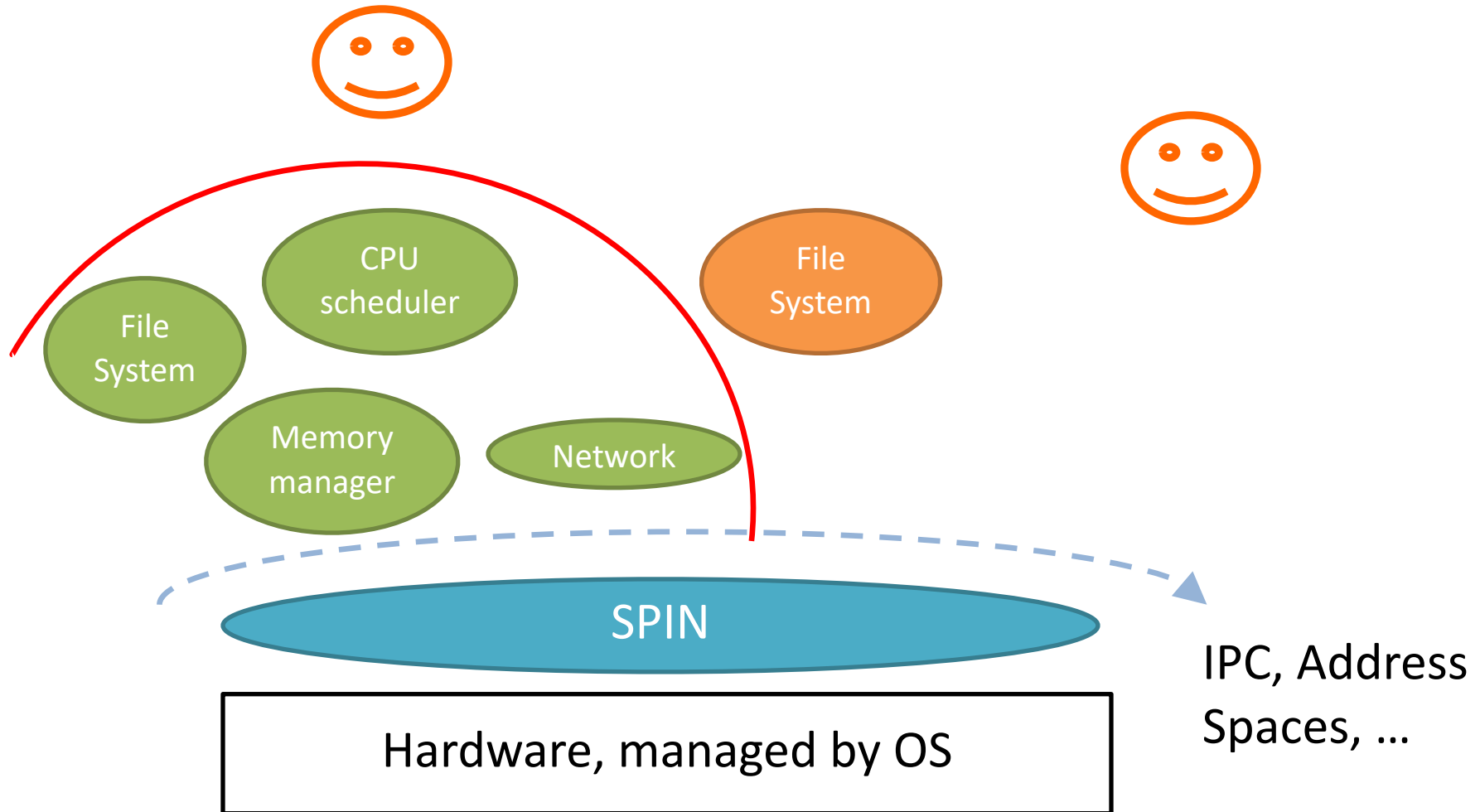
Spin



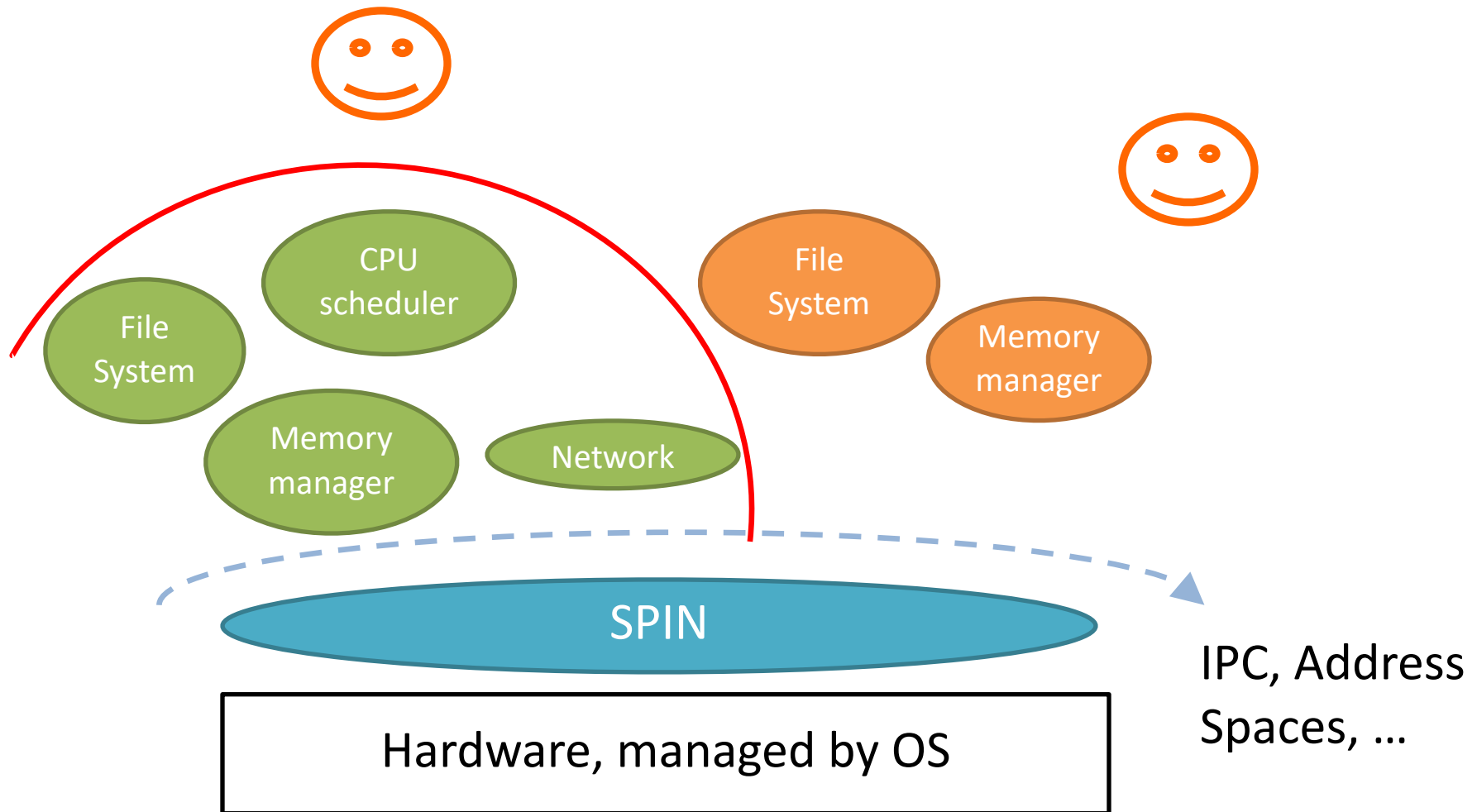
Spin



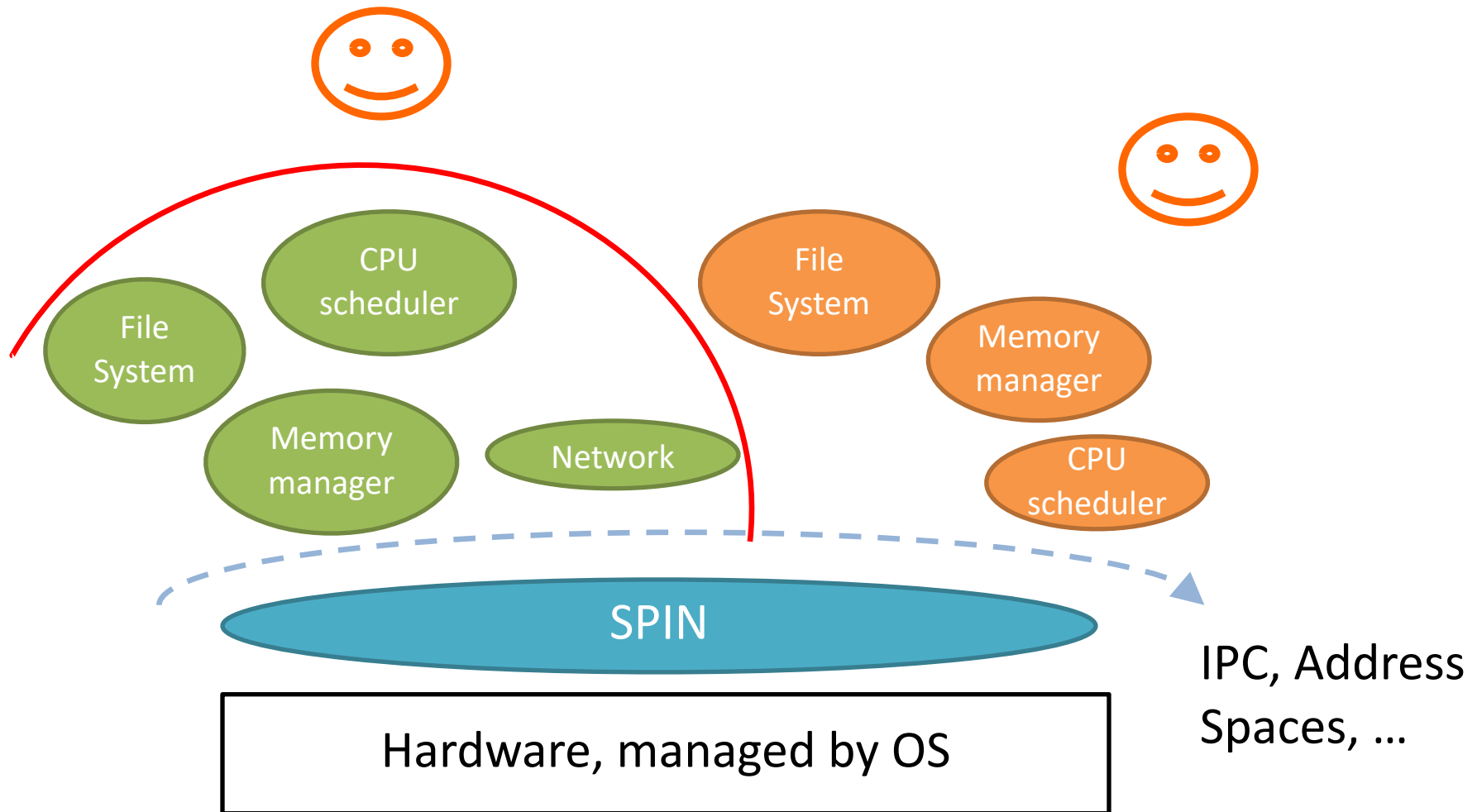
Spin



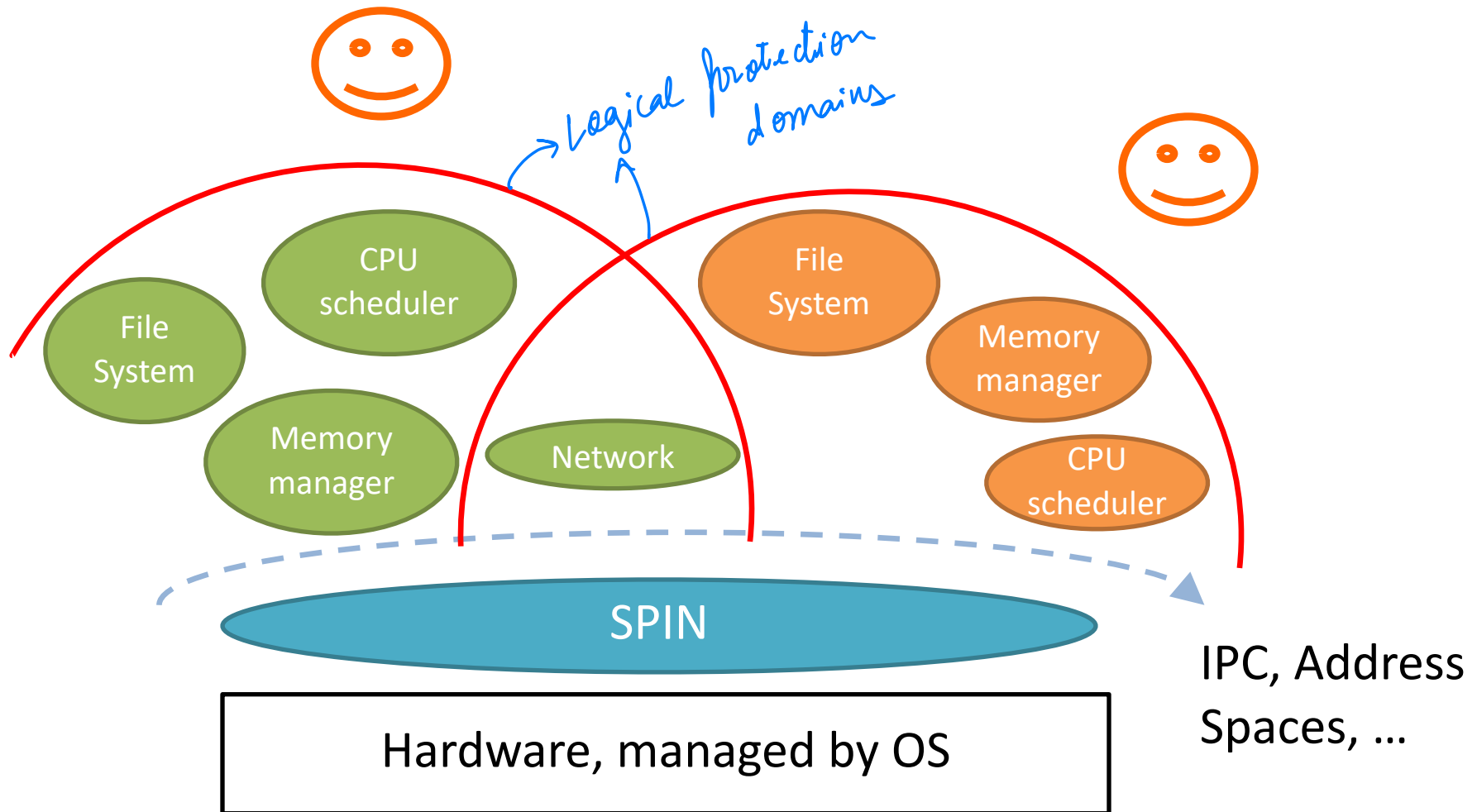
Spin



Spin



Spin



Spin Mechanisms for Events

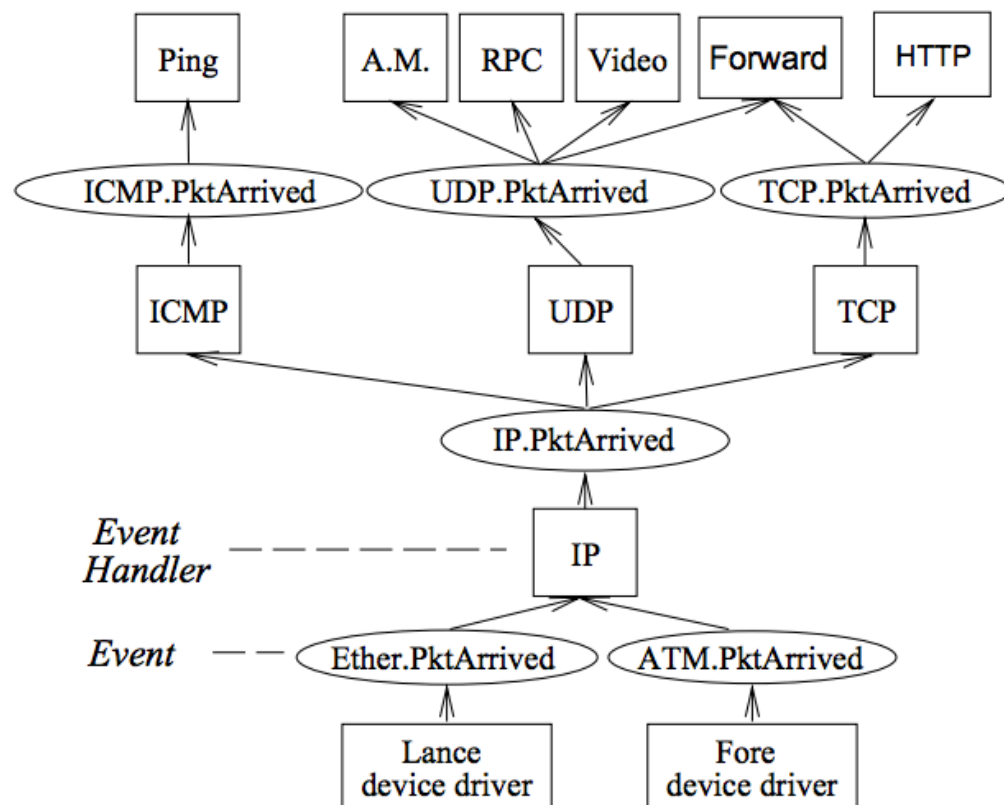
Spin Mechanisms for Events

- Spin extension model is based on events and handlers
 - Which provide for communication between the base and the extensions

Spin Mechanisms for Events

- Spin extension model is based on events and handlers
 - Which provide for communication between the base and the extensions
- Events are routed by the **Spin Dispatcher** to handlers
 - Handlers are typically extension code called as a procedure by the dispatcher
 - One-to-one, one-to-many or many-to-one
 - All handlers registered to an event are invoked
 - **Guards** may be used to control which handler is used

Event example



- Direct transfer from network to frame buffer
- Support of active networks
- In kernel handling of HTTP requests
- Support of Remote Procedure Call (RPC)
- Precursor to packet filters!

Figure 5: This figure shows a protocol stack that routes incoming network packets to application-specific endpoints within the kernel. Ovals represent events raised to route control to handlers, which are represented by boxes. Handlers implement the protocol corresponding to their label.

Core Services: Memory Management

- Memory management → *Only header files provided*
 - Physical address service
 - Allocate, deallocate, reclaim physical pages
 - Virtual address service
 - Allocate, deallocate capabilities for virtual addresses
 - Translation service
 - Add/remove mapping between virtual-physical addresses
 - Event handlers
 - Page fault, Access fault, Bad address
- Address space can be defined on top of the primitives
 - E.g., UNIX address space

Core Services: Memory Management

```

INTERFACE PhysAddr;

TYPE T <: REFANY; (* PhysAddr.T is opaque *)

PROCEDURE Allocate(size: Size; attrib: Attrib): T;
(* Allocate some physical memory with
   particular attributes. *)

PROCEDURE Deallocate(p: T);

PROCEDURE Reclaim(candidate: T): T;
(* Request to reclaim a candidate page.
   Clients may handle this event to
   nominate alternative candidates. *)

END PhysAddr.

```

```

INTERFACE VirtAddr;

TYPE T <: REFANY; (* VirtAddr.T is opaque *)

PROCEDURE Allocate(size: Size; attrib: Attrib): T;
PROCEDURE Deallocate(v: T);
END VirtAddr.

```

```

INTERFACE Translation;
IMPORT PhysAddr, VirtAddr;

TYPE T <: REFANY; (* Translation.T is opaque *)

PROCEDURE Create(): T;
PROCEDURE Destroy(context: T);
(* Create or destroy an addressing context *)

PROCEDURE AddMapping(context: T; v: VirtAddr.T;
                    p: PhysAddr.T; prot: Protection);
(* Add [v,p] into the named translation context
   with the specified protection. *)

PROCEDURE RemoveMapping(context: T; v: VirtAddr.T);

PROCEDURE ExamineMapping(context: T;
                        v: VirtAddr.T): Protection;

(* A few events raised during *)
(* illegal translations *)
PROCEDURE PageNotPresent(v: T);
PROCEDURE BadAddress(v: T);
PROCEDURE ProtectionFault(v: T);

END Translation.

```

Figure 3: *The interfaces for managing physical addresses, virtual addresses, and translations.*

Core Services: Thread Management

- Spin abstraction: strand
 - Similar to a thread in traditional OSs
 - But semantics/interfaces are defined by extension
- Event handlers
 - Block, unblock, checkpoint, resume
- Spin global scheduler
 - Interacts with extension thread packages (application-specific schedulers)

Core Services: Thread Management

```
INTERFACE Strand;  
  
TYPE T <: REFANY;  (* Strand.T is opaque *)  
  
PROCEDURE Block(s:T);  
(* Signal to a scheduler that s is not runnable. *)  
  
PROCEDURE Unblock(s: T);  
(* Signal to a scheduler that s is runnable. *)  
  
PROCEDURE Checkpoint(s: T);  
(* Signal that s is being descheduled and that it  
   should save any processor state required for  
   subsequent rescheduling. *)  
  
PROCEDURE Resume(s: T);  
(* Signal that s is being placed on a processor and  
   that it should reestablish any state saved during  
   a prior call to Checkpoint. *)  
  
END Strand.
```

Figure 4: *The Strand Interface.* This interface describes the scheduling events affecting control flow that can be raised within the kernel. Application-specific schedulers and thread packages install handlers on these events, which are raised on behalf of particular strands. A trusted thread package and scheduler provide default implementations of these operations, and ensure that extensions do not install handlers on strands for which they do not possess a capability.

Performance: Microbenchmark

- Competitors
 - DEC OSF/1: monolithic kernel
 - Mach 3.0: microkernel
- Protected communication

Operation	DEC OSF/1	Mach	<i>SPIN</i>
Protected in-kernel call	n/a	n/a	.13
System call	5	7	4
Cross-address space call	845	104	89

- Thread management

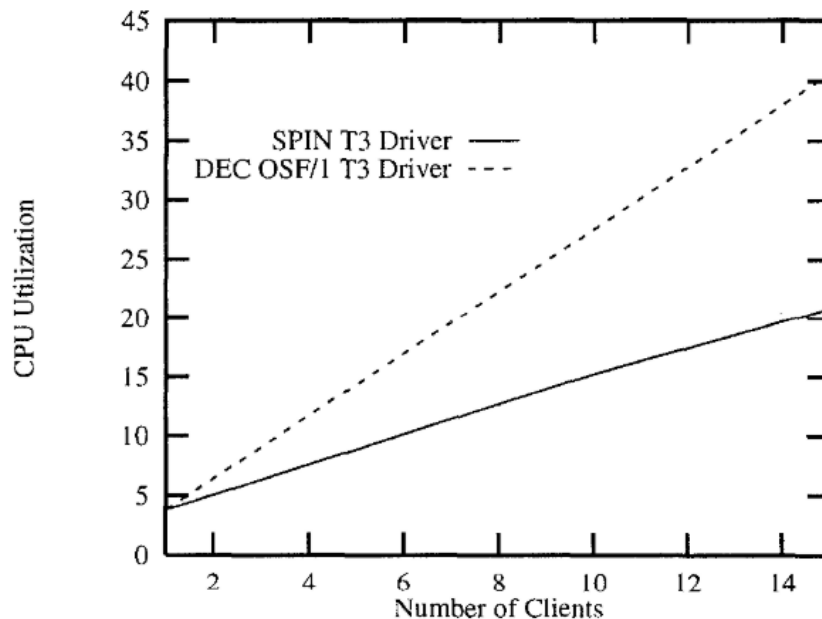
Operation	DEC OSF/1		Mach		<i>SPIN</i>		
	kernel	user	kernel	user	kernel	user	
						layered	integrated
Fork-Join	198	1230	101	338	22	262	111
Ping-Pong	21	264	71	115	17	159	85

Experiments: Networking

- Latency & Bandwidth

	Latency		Bandwidth	
	DEC OSF/1	<i>SPIN</i>	DEC OSF/1	<i>SPIN</i>
Ethernet	789	565	8.9	8.9
ATM	631	421	27.9	33

- Server utilization



- Substantial performance advantages even relative to a mature monolithic kernel
- Kernel extension v.s. user-level process
- Are you convinced?

We compare the performance of operations on three operating systems that run on the same platform: *SPIN* (V0.4 of August 1995), DEC OSF/1 V2.1 which is a monolithic operating system, and Mach 3.0 which is a microkernel. We collected our measurements on DEC Alpha 133MHz AXP 3000/400 workstations, which are rated at 74 SPECint 92. Each machine has 64 MBs of memory, a 512KB unified external cache, an HP C2247-300 1GB disk-drive, a 10Mb/sec Lance Ethernet interface, and a FORE TCA-100 155Mb/sec ATM adapter card connected to a FORE ASX-200 switch. The FORE cards use programmed I/O and can maximally deliver only about 53Mb/sec between a pair of hosts [Brustoloni & Bershad 93]. We avoid comparisons with operating systems running on different hardware as benchmarks tend to scale poorly for a variety of architectural reasons [Anderson et al. 91]. All measurements are taken while the operating systems run in single-user mode.

Scalability and the dispatcher

SPIN's event dispatcher matches event raisers to handlers. Since every procedure in the system is effectively an event, the latency of the dispatcher is critical. As mentioned, in the case of a single synchronous handler, an event raise is implemented as a procedure call from the raiser to the handler. In other cases, such as when there are many handlers registered for a particular event, the dispatcher takes a more active role in event delivery. For each guard/handler pair installed on an event, the dispatcher evaluates the guard and, if true, invokes the handler. Consequently, dispatcher latency depends on the number and complexity of the guards, and the number of event handlers ultimately invoked.

Impact of automatic storage management

An extensible system cannot depend on the correctness of unprivileged clients for its memory integrity. As previously mentioned, memory management schemes that allow extensions to return objects to the system heap are unsafe because a rogue client can violate the type system by retaining a reference to a freed object. *SPIN* uses a trace-based, mostly-copying, garbage collector [Bartlett 88] to safely reclaim memory resources. The collector serves as a safety net for untrusted extensions, and ensures that resources released by an extension, either through inaction or as a result of premature termination, are eventually reclaimed.

Clients that allocate large amounts of memory can trigger frequent garbage collections with adverse global effects. In practice, this is less of a problem than might be expected because *SPIN* and its extensions avoid allocation on fast paths. For example, none of the measure-

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
- Exokernel and SPIN try to achieve all three!

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

Discussions

- SPIN vs. Exokernel
 - Spin: kernel extensions safely specialize OS services through PL support
 - Exo-kernel: securely expose hardware resources, decoupling protection from management

Discussions

- SPIN vs. Exokernel
 - Spin: kernel extensions safely specialize OS services through PL support
 - Exo-kernel: securely expose hardware resources, decoupling protection from management
- Microkernel vs. Exokernel
 - Microkernel: focuses on IPC between processes
 - IPC and process management abstractions are fixed and cannot be changed
 - The intent is that most OS functionality (e.g. file system, network protocols) be in user-level server processes
 - Exokernel: expects that most OS functionality takes the form of LibOS linked with each user process
 - Much lower interfaces exported; less border crossing possible due to downloadable code

Discussions

- SPIN vs. Exokernel
 - Spin: kernel extensions safely specialize OS services through PL support
 - Exo-kernel: securely expose hardware resources, decoupling protection from management
- Microkernel vs. Exokernel
 - Microkernel: focuses on IPC between processes
 - IPC and process management abstractions are fixed and cannot be changed
 - The intent is that most OS functionality (e.g. file system, network protocols) be in user-level server processes
 - Exokernel: expects that most OS functionality takes the form of LibOS linked with each user process
 - Much lower interfaces exported; less border crossing possible due to downloadable code
- What about VMs?
 - VMs emulate full machine and also typically run another monolithic kernel inside of it

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