

# Operating System Structure

- Broad goals of an operating system
  - Make it easier for a user to use the system
  - Manage the system resources efficiently among multiple users
- What is a “resource”?
  - Any logical component that a OS manages
  - A user has to request the use of the resource from the OS, cannot directly access it
  - Ex. CPU, memory, devices, files, ...
  - Allows the details of the hardware operations and management to be hidden from the user

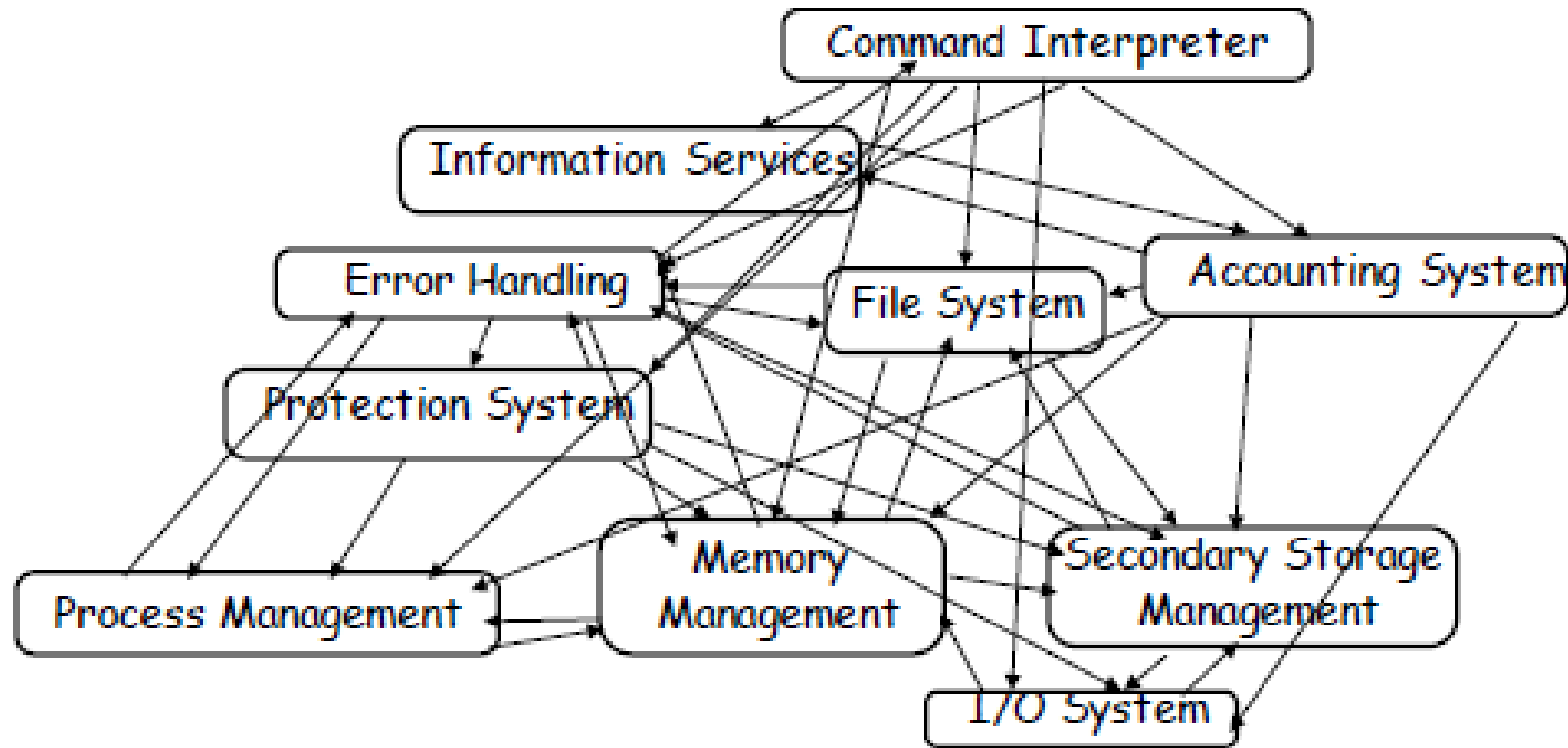
- Resource management can be complex
  - Allocation and deallocation of limited resources among multiple processes
  - Some resources may be shared between processes
    - Requires mechanisms for exclusive use
      - Software, hardware, software+hardware?
  - Reclaiming unused resources
  - Dependency between resources
- Resource manager can be a combination of complex, interacting programs

# Functionalities of an OS

- Process and associated resource management
  - Provide illusions of multiple virtual machines
- Memory management
  - Allocation to processes
  - Reclaiming unused memory
  - Protection from unauthorized access
  - Mechanisms for sharing
- Device Management
  - Provides controlling functions to many devices
- File Management
  - Provides organization and storage abstractions

- Communication
  - Between processes
  - Between machines
- Protection and Security
- Accounting
- Error detection and reporting
- User interface

A collection of interacting programs provide these services



How do we put them all together?

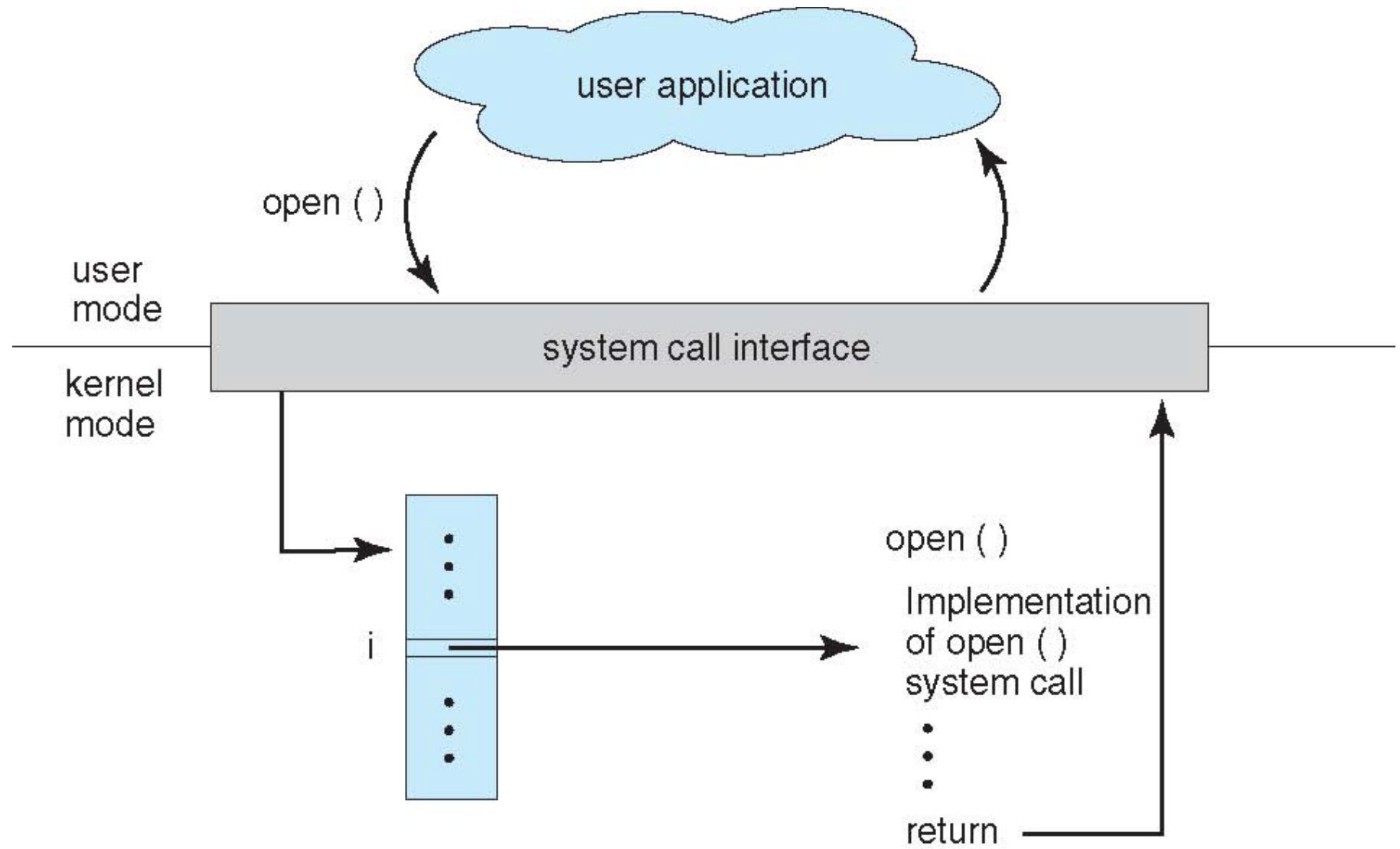
# OS Organization

- Kernel
  - A protected part of the OS, user programs cannot access it directly
- User programs
  - Application programs/services built using the services provided by the kernel
- Why this division needed?
  - To prevent direct access to devices and memory for protection
- How is protection implemented?
  - Uses hardware supported special privileged instructions that can only be executed in a privileged mode
  - Kernel programs are allowed to execute them
  - User programs are not allowed to execute them

- So how can user programs do something protected
  - No direct way, need to ask the OS
  - System Calls – special functions that user programs can call to request kernel services
- How do system calls work?
  - Set of known system calls, each with a specific id (number)
  - Calling function puts all parameters in predefined places (registers)
  - Calling function puts the system call id also in a register
  - Makes a special call (same for all system calls; ex syscall)

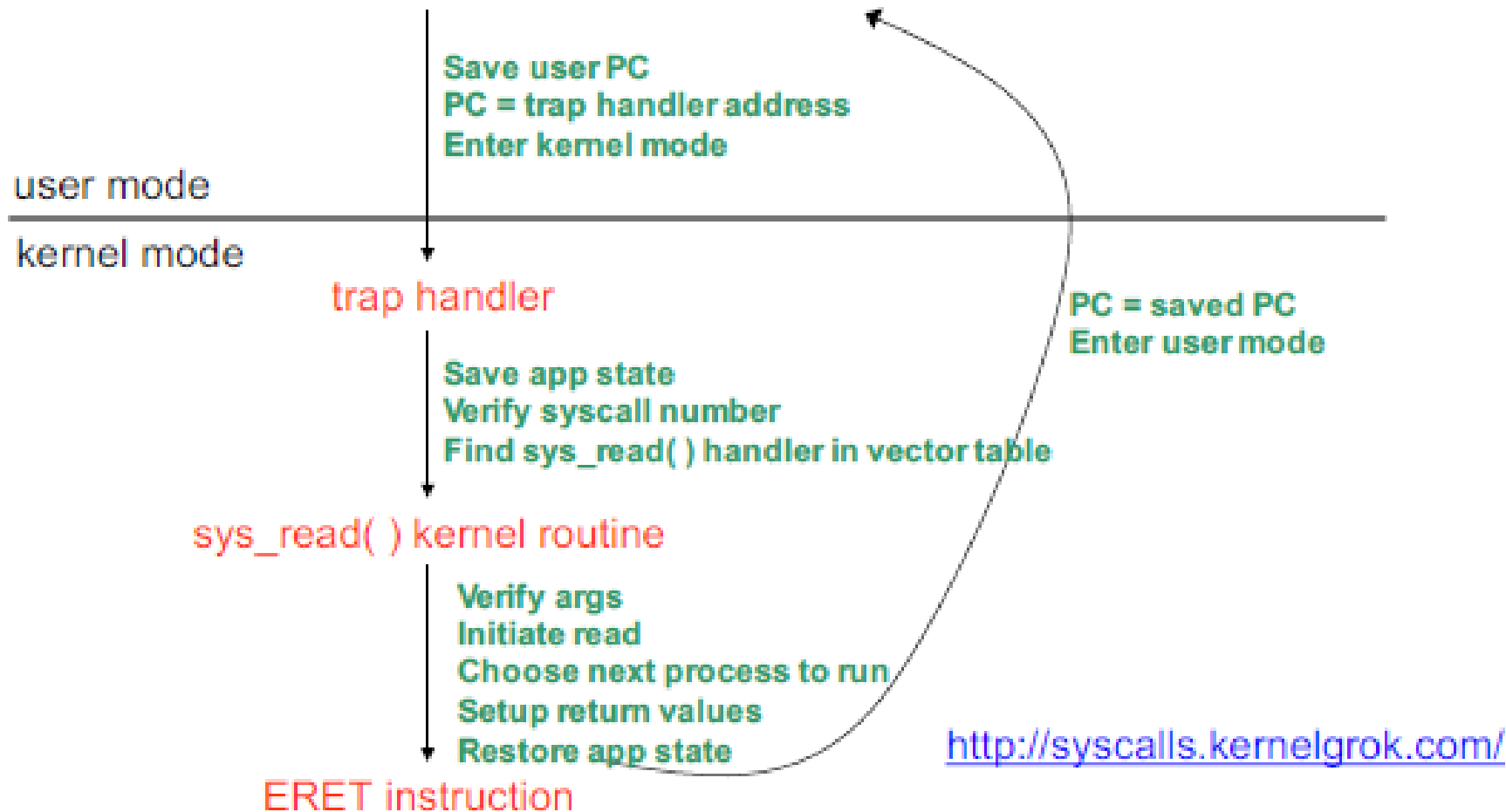


- The special call
  - Saves current PC
  - Changes execution mode to privileged
  - Sets PC to a handler function
- Handler function
  - Checks the system call id
  - Verifies the arguments
  - Jumps to the specific system call handler
- After the system call is executed, a special return instruction is executed that sets the PC to the return address in the user program code
  - Also sets execution mode back to non-privileged.



# Example

Firefox: read(int fileDescriptor, void \*buffer, int numBytes)

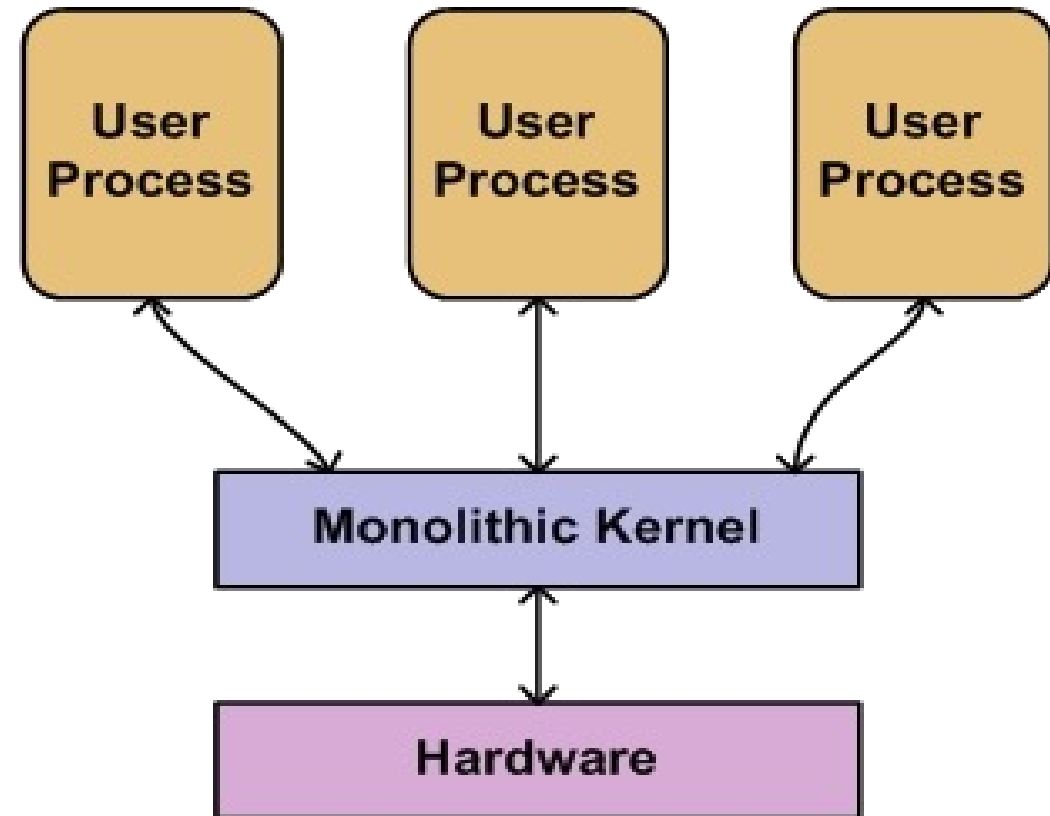


# Structure of an OS

- General design goals of an OS
  - User goals – should be easy to use, easy to learn, reliable, secure, fast
  - System goals – should be easy to design, easy to implement, easy to maintain, reliable, flexible, secure, fast
- Given the above goals
  - How do we organize all the programs that make up an OS?
  - How do we decide what goes in the kernel and what goes in user mode?
  - Different approaches

# Monolithic OS

- Put everything in the kernel
- Only application programs in the user space
- Kernel handles all system tasks, applications have no control
- Example: Windows 9x, FreeBSD, Unix variants



Ref: Kaashoek et al.

- Advantages

- Lower overhead possible between multiple kernel modules due to direct interaction
- Strong protection from user programs
- Application programs can get all services from one place

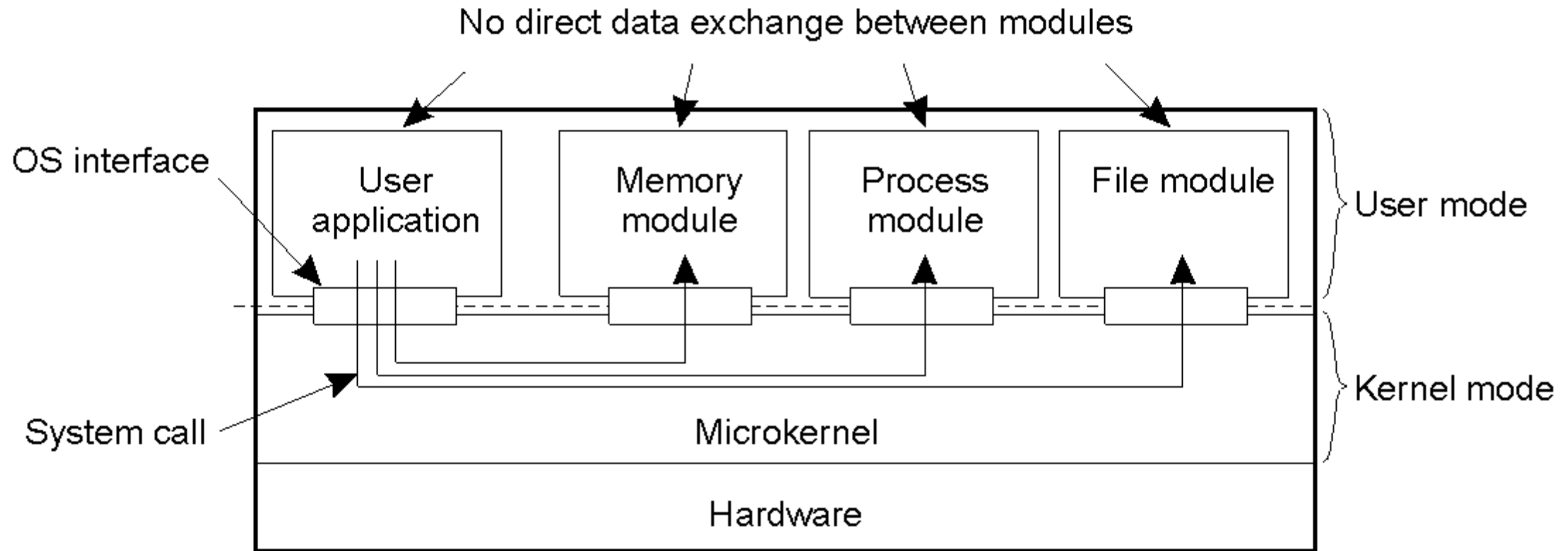
- Disadvantages

- Bulky code that is hard to implement and maintain
- Hard to change anything or add new components/services
- Slower, as system calls are made for everything, and system calls are costly
- Little isolation between different OS modules

# Microkernel based OS

- Only minimal services implemented in the kernel
  - Example: CPU scheduling, interprocess communication, memory allocation/deallocation
- Rest are implemented in user space as needed
  - Example: Memory management, File system management, Networking,...
- Example: L4 Microkernel, MACH (precursor of MacOS), QNX

# Microkernel based OS





# Microkernel based OS

- Advantages

- Smaller code that can be easily implemented, tested, and maintained
- Easier to modify/add/remove services
- Easier to port OS to new architectures
- More reliable (less code is running in kernel mode)
- More secure

- Disadvantages:

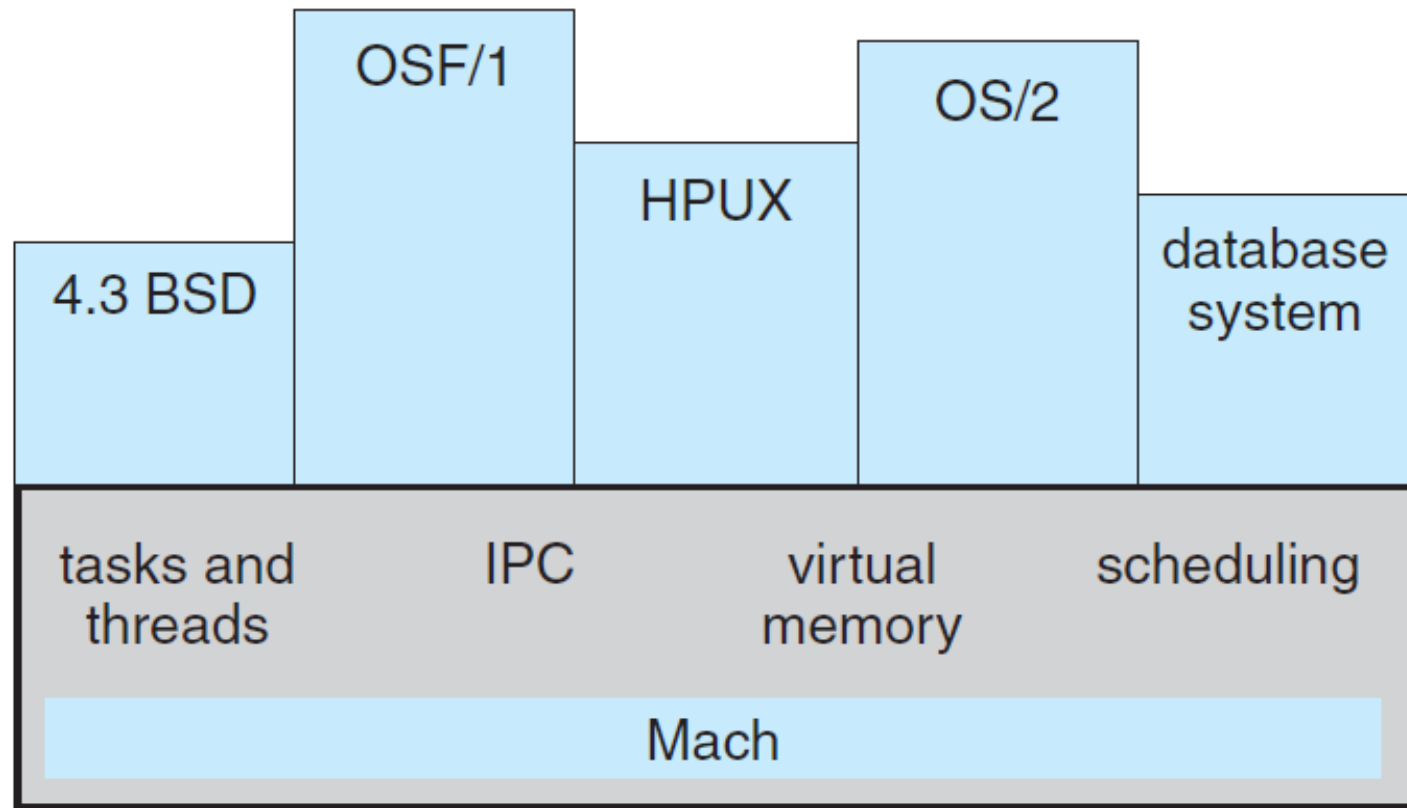
- Too much user space to kernel space communication

# Example: Mach

- Project at CMU in 80's
- Mach 2
  - Many improvements over an earlier OS, Accent, developed in CMU
    - Threads, better IPC, multiprocessor support, virtual memory support
  - Native support for 4.2/4.3 BSD code
    - Large kernel, but fully compatible with BSD, which was a DARPA requirement
- Mach 3 (1990-94)
  - Removes all BSD code from kernel and puts it in user space
  - User level OS emulator for BSD Unix

# Basic Approach

- Has a very thin kernel, with most OS functionality moved out of the kernel
- Object oriented approach, however, objects cannot directly call functions on other objects, they have to communicate by message passing
  - Individual, well-defined components (in effect, subsystems) communicate with one another by means of messages
- Mach provides a very thin kernel, most OS functionalities are built on top of it in user space
  - Sort of a kernel-within-a-kernel
  - Several OS implementations exist that use Mach at the bottom
    - BSD implementation
    - XNU, an OS implemented on top of Mach



- In Mach, everything is implemented as its own object
- Processes (which Mach calls tasks), threads, and virtual memory are objects, each with its own properties
- Mach implements object-to-object communication by means of message passing
- Unlike other architecture, Mach objects cannot directly invoke or call on one another, rather they are required to pass messages

- The source object sends a message, which is queued by the target object until it can be processed and handled
- Similarly, the message processing may produce a reply, which is sent back by means of a separate message
- Messages are delivered reliably in a FIFO manner
- The content of the message is entirely up to the sender and the receiver to negotiate

# Mach Kernel Abstractions

- A small set of abstractions of that is simple but powerful
  - Tasks
    - Provides an environment for program execution
    - Units of resource ownership
    - Each task consists of a virtual address space, a port right namespace, and one or more threads (Similar to a process.).
    - Allocation of resources to individual threads or groups
    - The task itself does no execution, it is a framework for running threads
  - Threads
    - Basic unit of execution/scheduling
    - All threads of a task share the same virtual address space and port rights
    - Multiple threads from one task can execute simultaneously

# Mach Kernel Abstractions

- Address space
  - In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
- Memory objects
  - The internal units of memory management.
  - Memory objects include named entries and regions; they are representations of potentially persistent data that may be mapped into address spaces
  - Provides for allocating/deallocating memory to tasks
  - Can specify specific access rights (like read-only) on parts of the address space, or which parts should to be inherited if a task is cloned



# Mach Kernel Abstractions

- Ports

- Secure, simplex communication channels (message queues)
- There is send/receive access rights defined for each port
- Tasks, threads, and other resources are associated with ports
- Tasks and threads can also have access rights to task ports and thread ports of other tasks and threads
- Any number of tasks/threads can have send rights on a port, but at a time only one task/thread can have receive rights
- Access rights can be passed from one task (thread) to another also through the ports (again by sending message)
- Access rights to a task/thread port allows calling Mach functions on that port
- Port have unique names
- Network-wide name service to look up port
  - Allows for messages to be passed over the network to another machines
- Messages are data streams with header and content, format known to sender and receiver

# Mach Kernel Abstractions

- IPC

- Message queues
- Remote procedure calls
- Notifications
- Semaphores
- Lock sets

- Time

- Clocks, timers, and waiting.

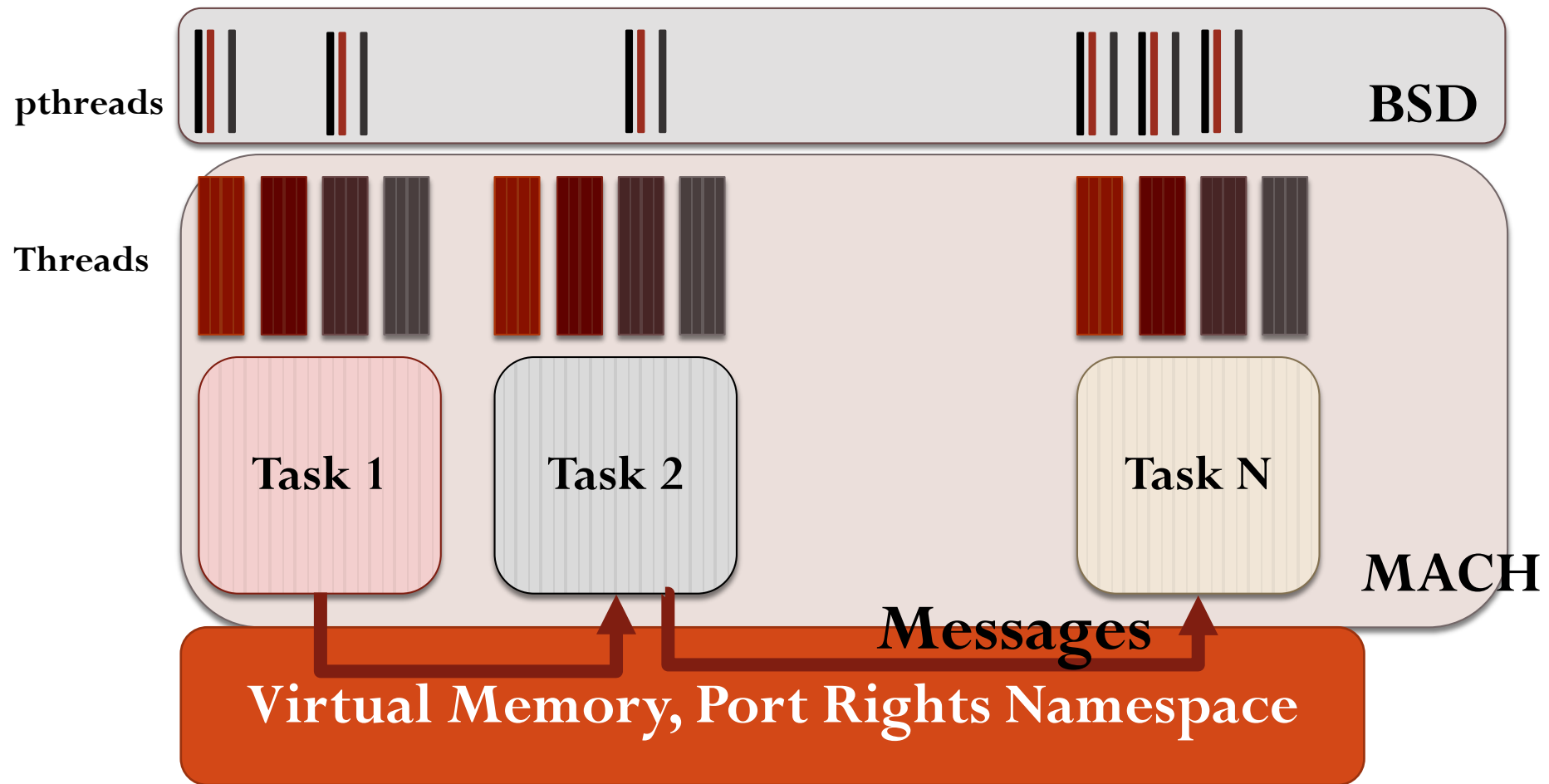
# Mach Interface

- Set of calls to access Mach services

- Bootstrap
- IPC
- Messaging
- Port manipulation
- Virtual memory
- Tasks and threads
- Scheduling
- Processor
- Device

<https://www.gnu.org/software/hurd/gnumach-doc/>

<https://web.mit.edu/darwin/src/modules/xnu/osfmk/man/>

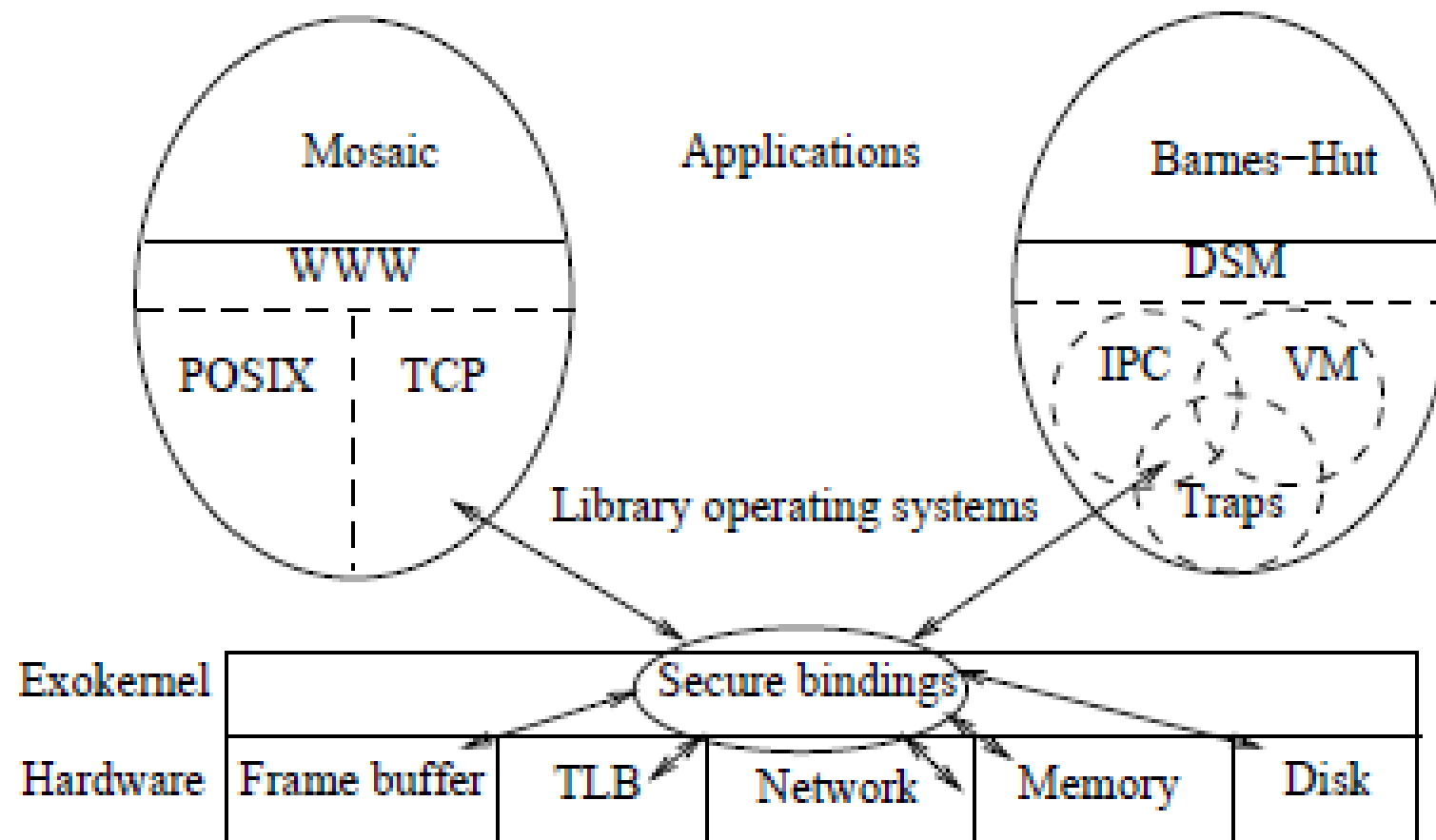


# Exokernel Architecture

- Philosophy: *“Applications know better than operating systems what the goal of their resource management decisions should be and therefore, they should be given as much control as possible over those decisions”*
- Traditional implementations have hardcoded abstractions
  - Processes, files, address spaces, IPC,...
- Different applications may have different needs
  - Hard to do domain specific optimizations
  - Examples
    - Applications may know their file access patterns but internal file caching policies in filesystem implementations may be different
    - Data access patterns may not match page replacement strategies

- Separate out management and protection
  - **Management:** Allocate resources based on the requirement
    - Example: An application asks for a memory block
    - However, read-write strategy over that memory block will be decided by the application
    - Done primarily by application
  - **Protection:** Ensures no conflict in resource allocation
    - Maintain ownership of resources
    - Example: Do not allocate a memory block if it is already allocated to another application
    - Done by Exokernel
    - Management done only for what is needed for protection

- Approach
  - Multiplex and Export hardware resources with low level primitives
    - Interrupts, timers, device I/O, physical memory, TLB, ...
  - Library OS's implement the higher level abstractions using these primitives
- Three important tasks
  - *Secure binding*: allows Library OS's to securely bind to machine resources'
  - *Visible revocation*: allows Library OS's to participate in resource revocation protocols
  - *Abort*: allows Exokernel to break bindings of uncooperative Library OS's by force



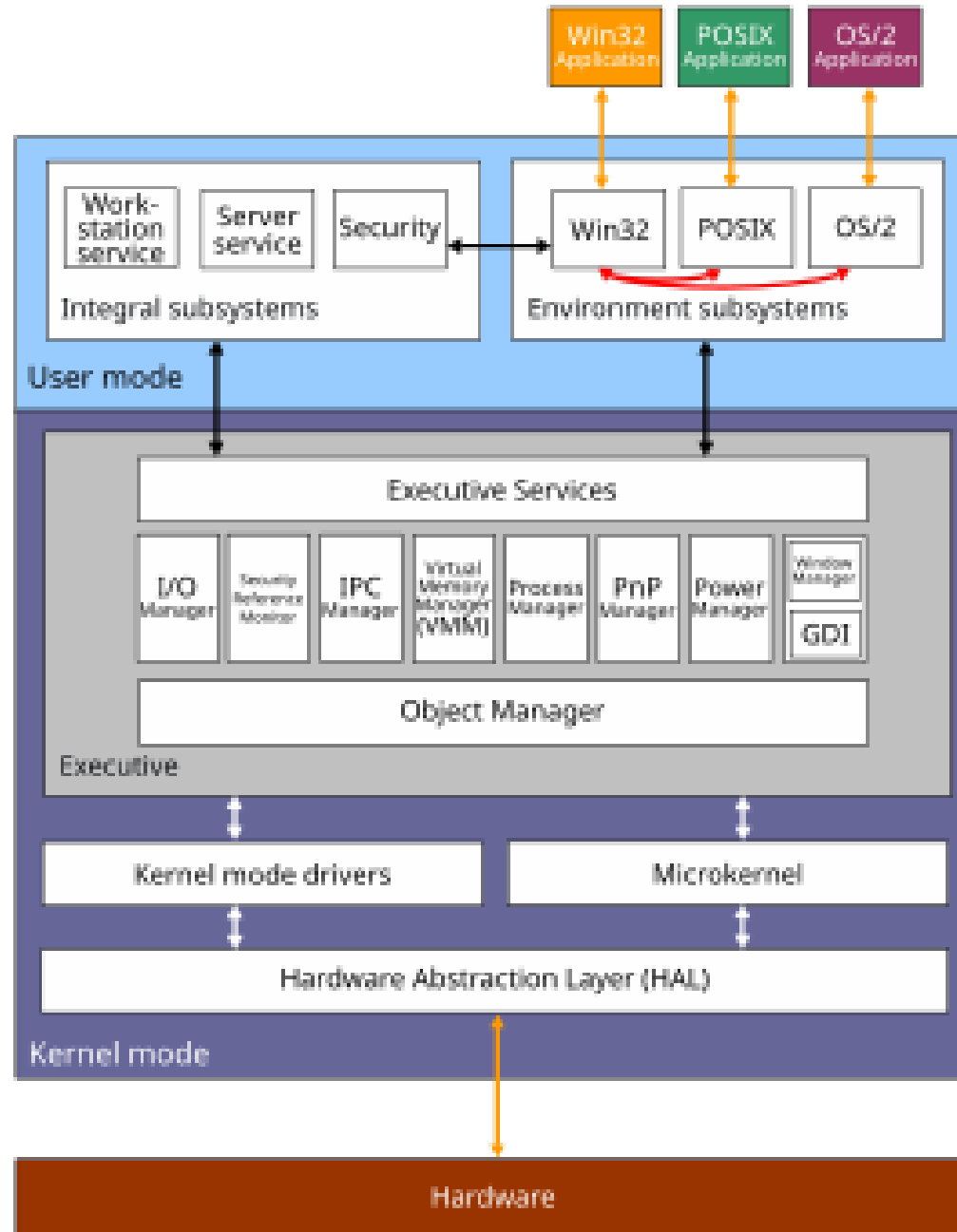


- An example
  - Library asks for physical memory from Exokernel
  - Exokernel allots the physical memory
    - Same memory is not allotted to two Library OSs
  - Library OS implements its own virtual memory system using it
  - When a TLB miss happens, Library OS finds the mapping using its virtual memory implementation
  - Asks Exokernel to put it in TLB
  - Exokernel checks if the physical memory in the mapping is in the range allocated to that Library OS
  - Add if yes, reject if no

# Hybrid Systems

- Attempts to combine monolithic and microkernel approaches
- Not a very standard term
- Examples
  - Linux is mainly monolithic, but allows Loadable Kernel Module for adding additional functionalities to the kernel easily
  - Windows is monolithic, but design inside has clear separation between core kernel services and executive services on top of it
  - XNU has Mach-based microkernel and BSD (and some other things) on top of it, all in kernel mode

# Windows



# XNU

User Mode	Application Enviroments Common Services Driver Kit	U s e r  S p a c e
K E R N E L	FreeBSD Filesystems, Networking, BSD Sockets, BSD Libraries, POSIX Thread Support OSFMK 7.3 IPC, Virtual Memory, Protected Memory, Scheduling, Preemptive Multitasking, Real-Time Support, Console I/O	X N U