Operating System Architecture

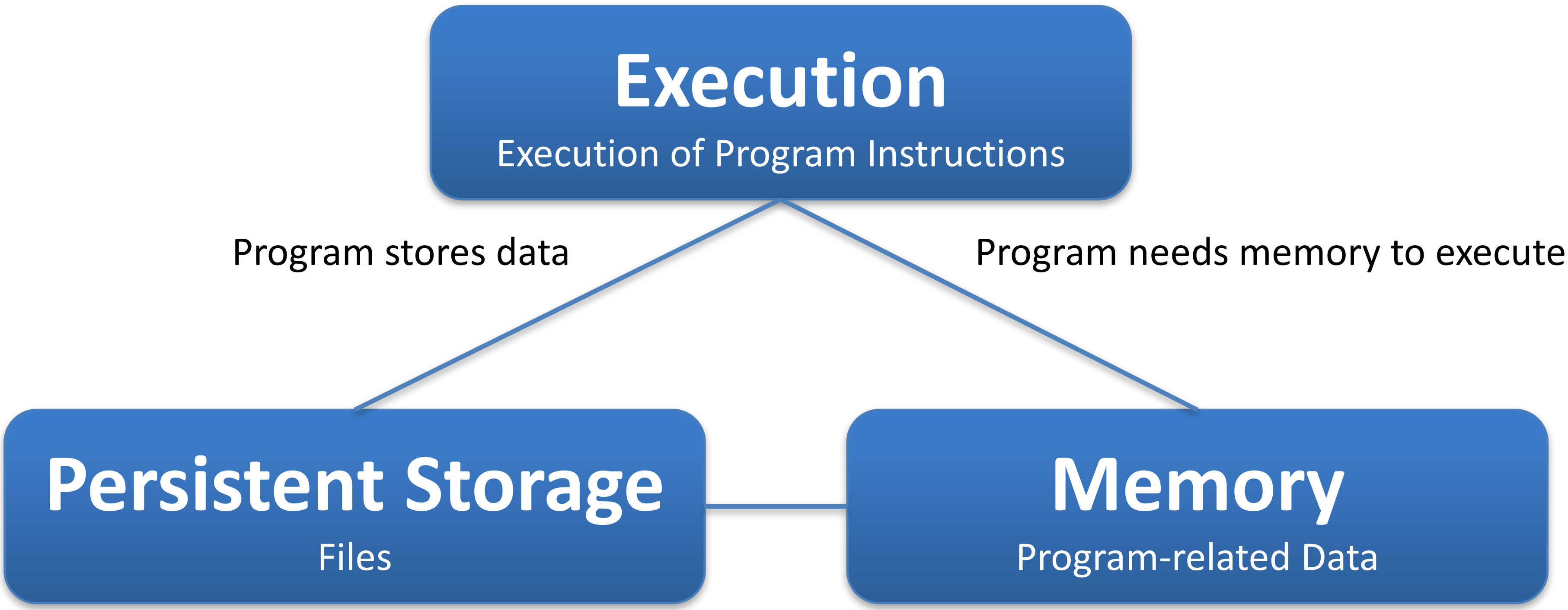
CS3026 Operating Systems

Lecture 03

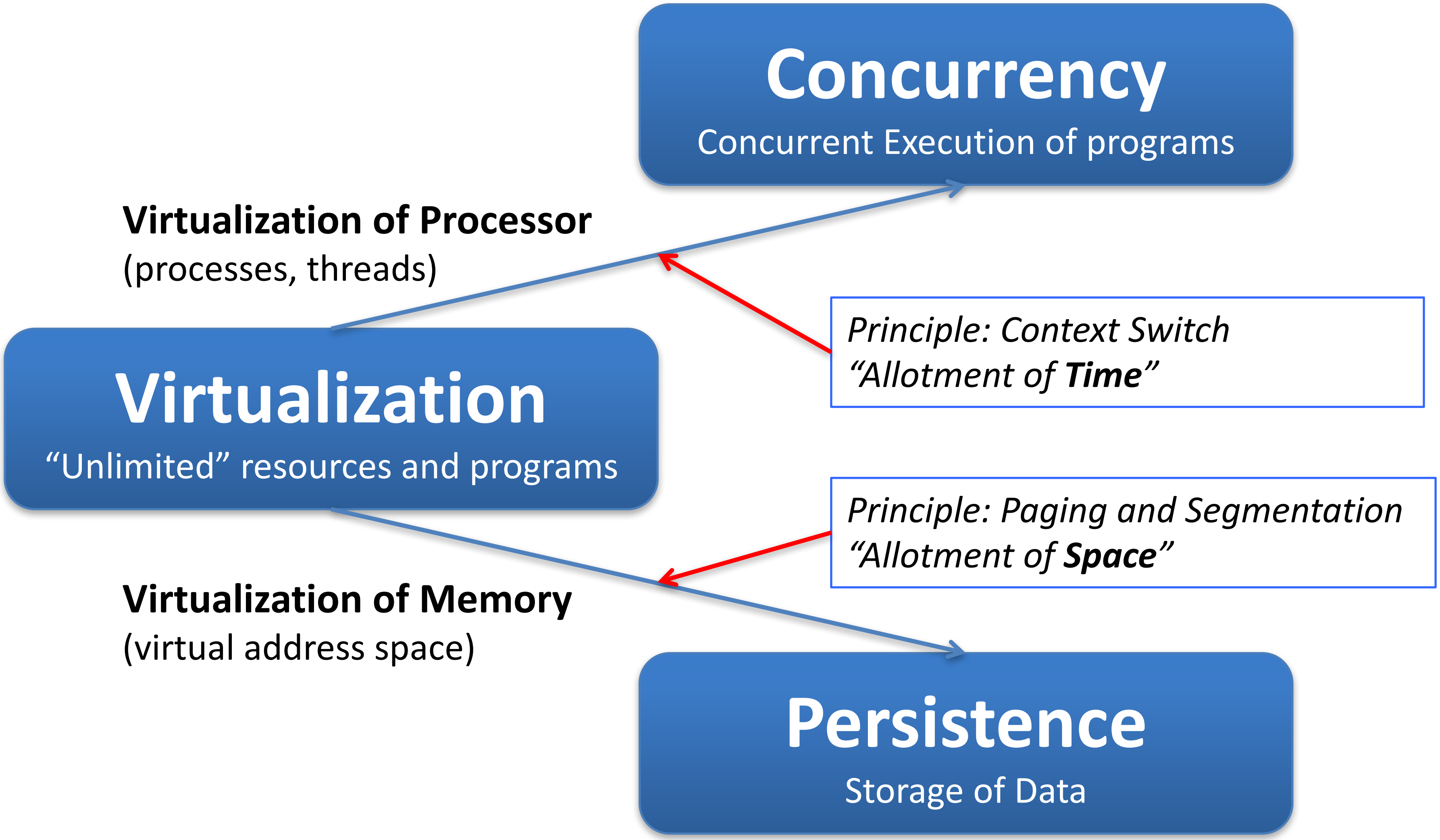
The Role of an Operating System

- Service provider
  - Provide a set of services to system users
- Resource allocator
  - Exploit the hardware resources of one or more processors and allocate it to user programs
- Control program
  - Control the execution of programs and operations of I/O devices
    - interrupt them to send/receive data via I/O or to re-allocate hardware resources to other user programs
- Protection and Security
  - Protect multiple programs running from each other
  - Secure user access to data and define ownership of files and processes

Operating System Functions



Core Concepts



Operating Systems

	Execution	Memory	Storage
Virtualization	Process, thread Context switch Process Control Block Swapping	Virtual memory management Segmentation Free space management Paging Page table TLB	Storage volumes File system
Concurrency	Mutual exclusion Locking Condition variables Semaphores Deadlock	Shared memory	Locking
Persistence			Files and directories I/O Devices Hard disk

Operating System Structure

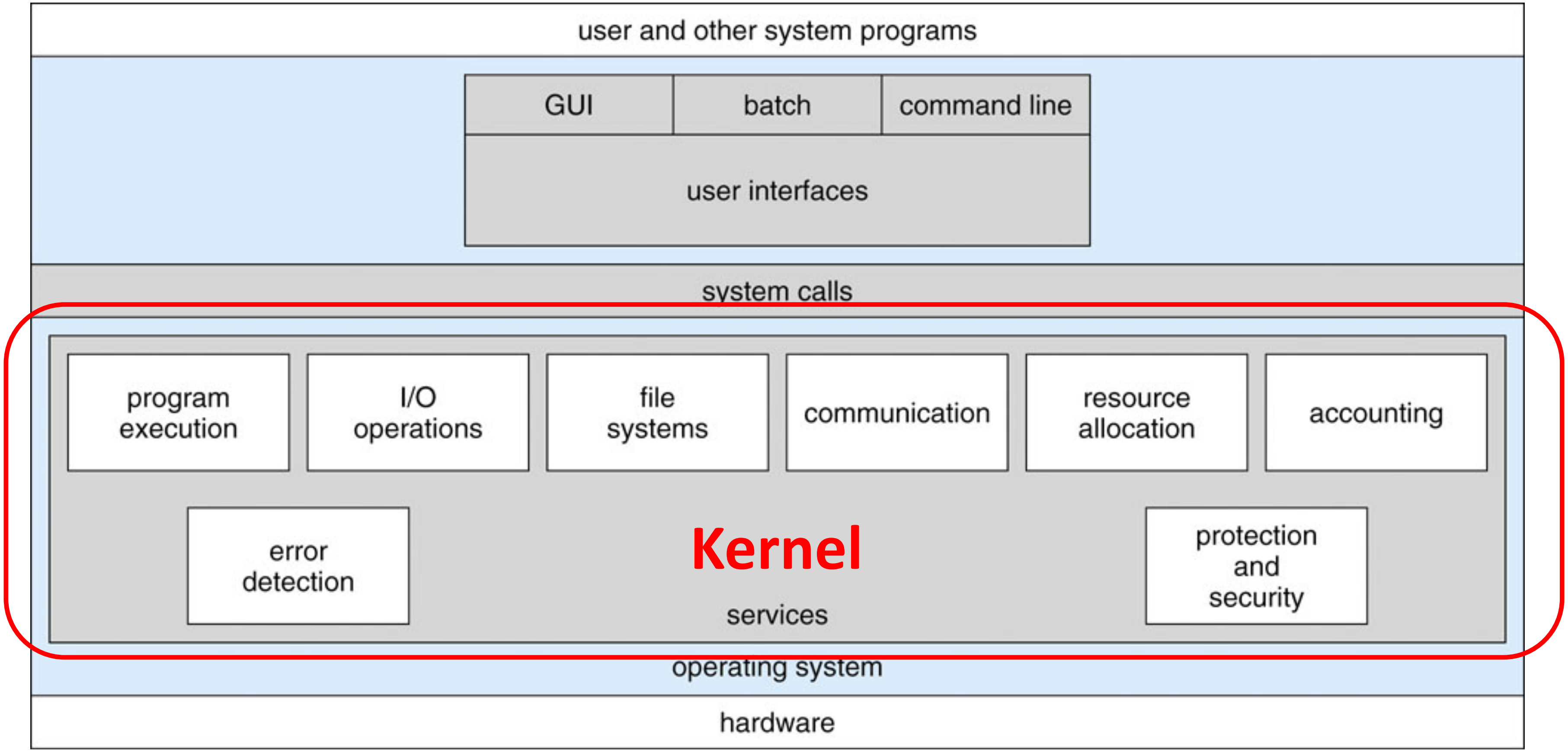
The Kernel

Protecting the Operating System
Modes of Operation

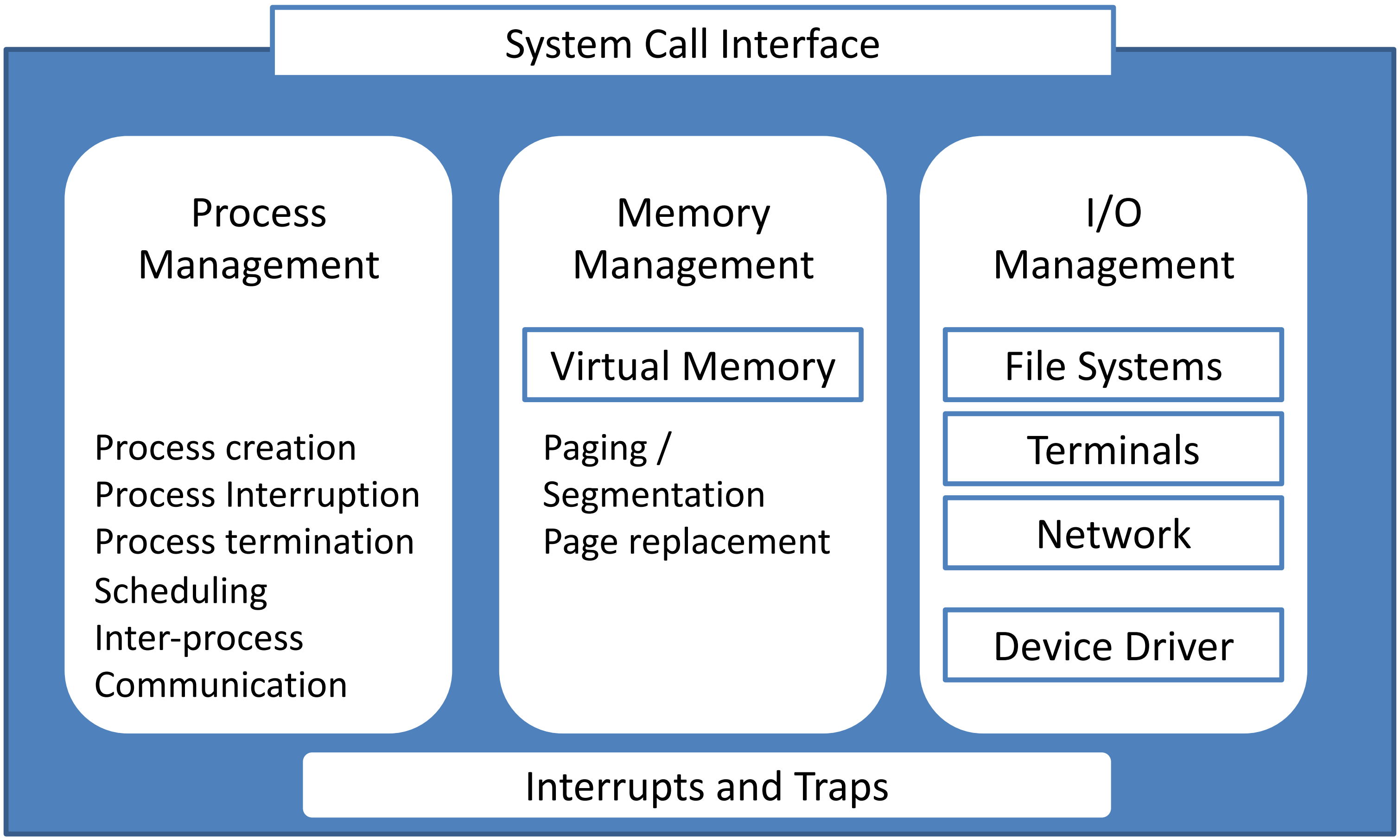
- User Mode
  - User programs execute in ***user mode***
  - Certain areas are protected from user access
  - Certain instructions may not be executed

- Kernel Mode
  - Operating system executes in ***kernel mode***
  - Privileged instructions may be executed
  - Protected areas of memory may be accessed

Operating System



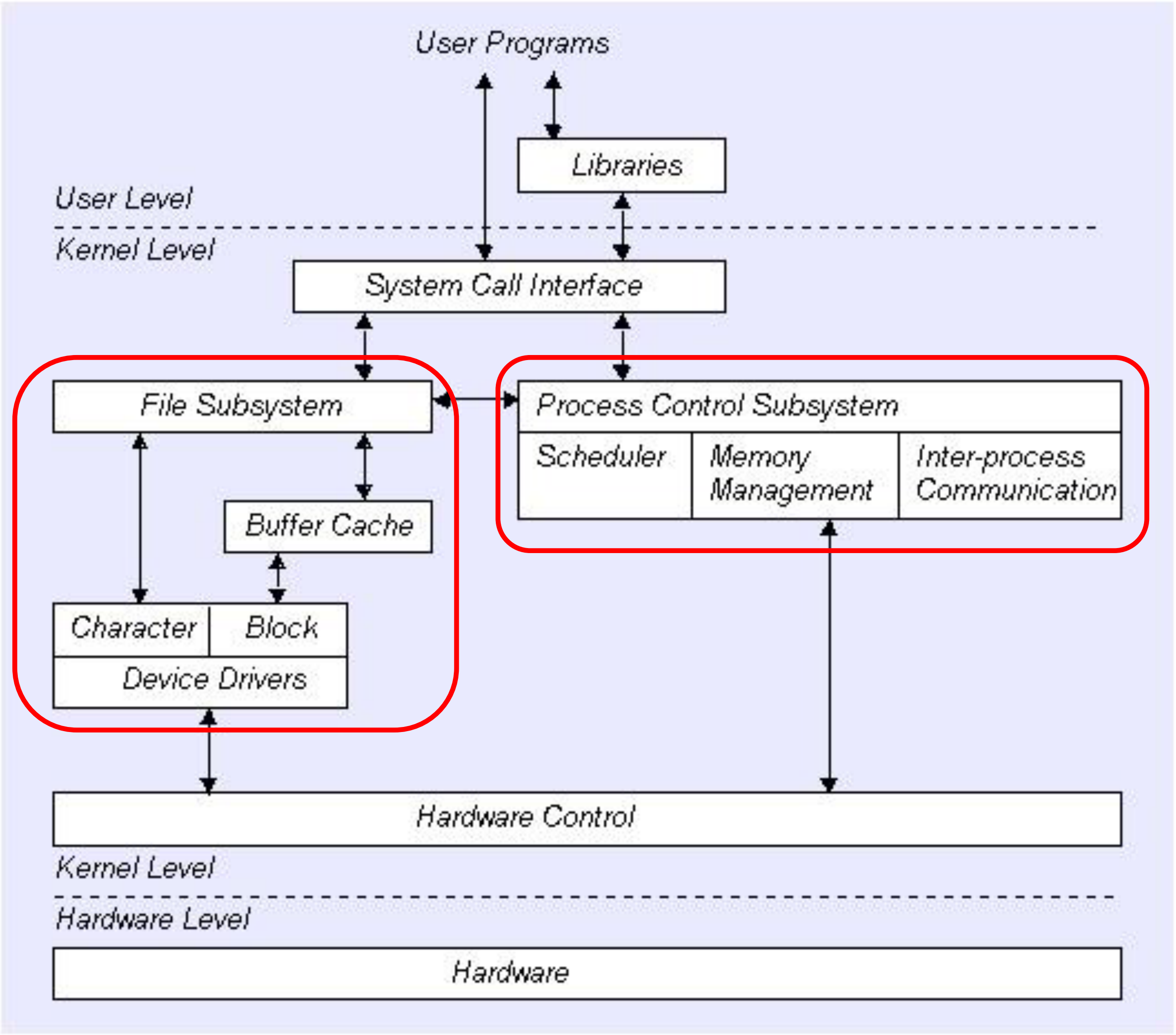
Basic Components of Unix Kernel



Utilising Hardware Resources

- Hardware supports
  - Basic instruction execution
  - Interrupt handling
  - Basic memory addressing mechanisms
  - User / kernel mode operation for protecting resources
- Operating system manages software constructs based on these hardware services
  - Process management
  - Virtual memory management
  - File storage and communication management

Traditional Unix Kernel



System Calls

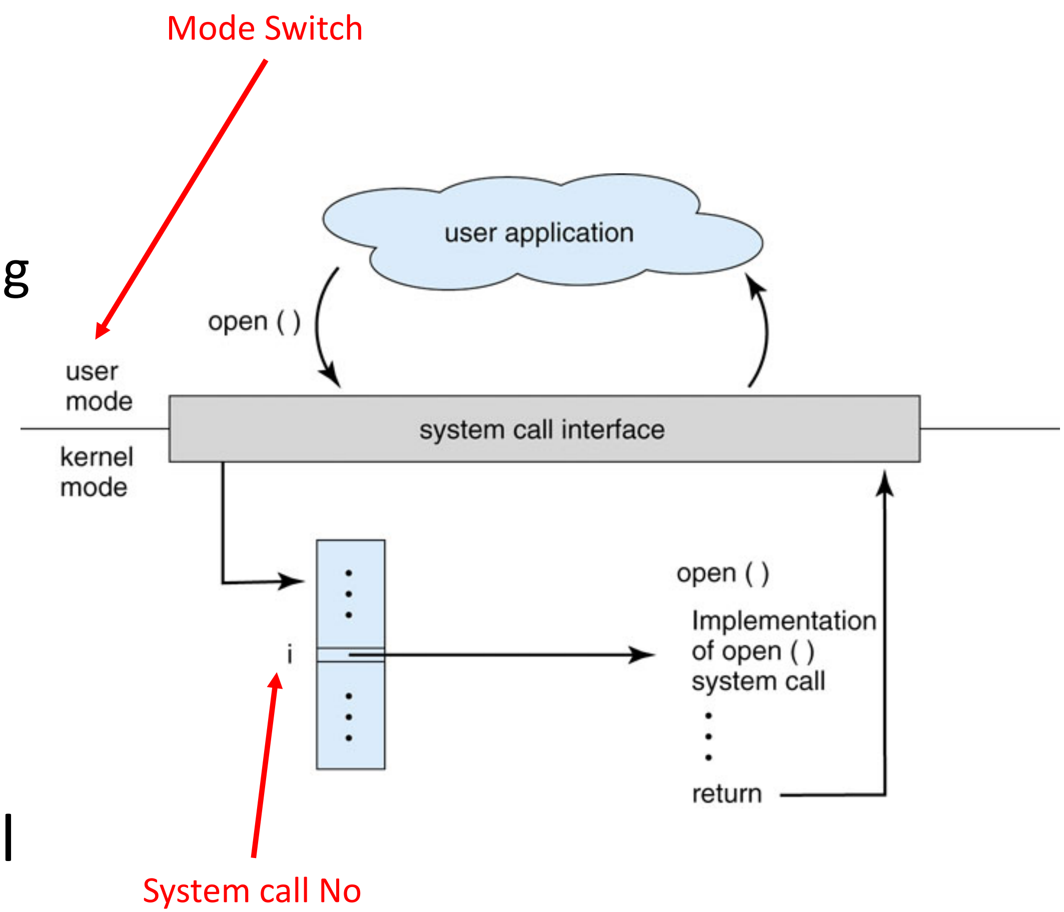
Calling Operating System Functions

System Calls

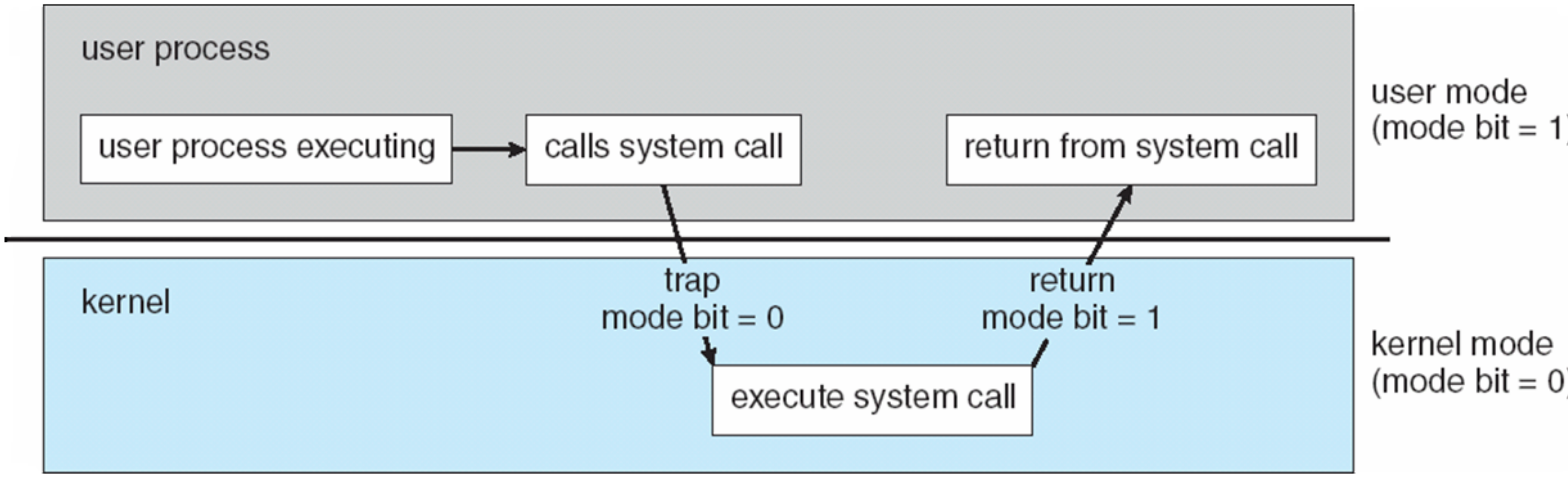
- System calls are the only entry point into the kernel
- Categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication
- System calls are executed in ***kernel mode***

System Calls

- Interface between a program and the operating system kernel
  - Provide access to operating system services
- Is an explicit request to the kernel made via a software interrupt
- Executed in ***kernel mode***
  - Requires a mode switch
- Each system call is identified by a system call number



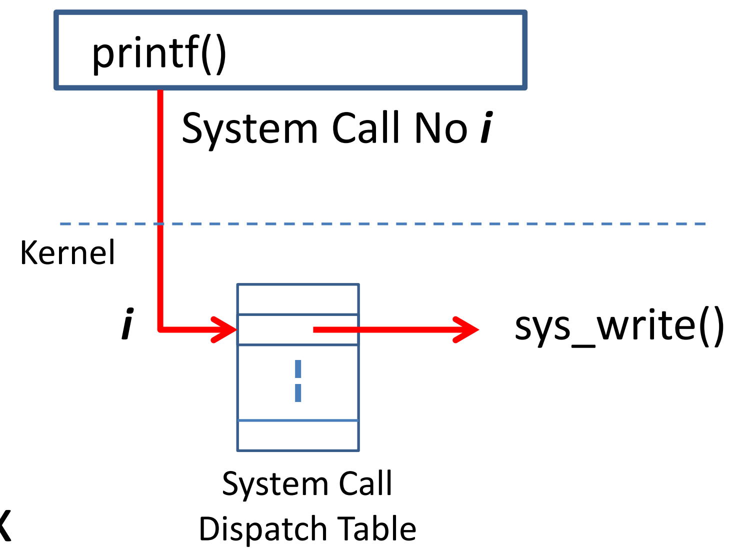
Software Interrupt for System Calls



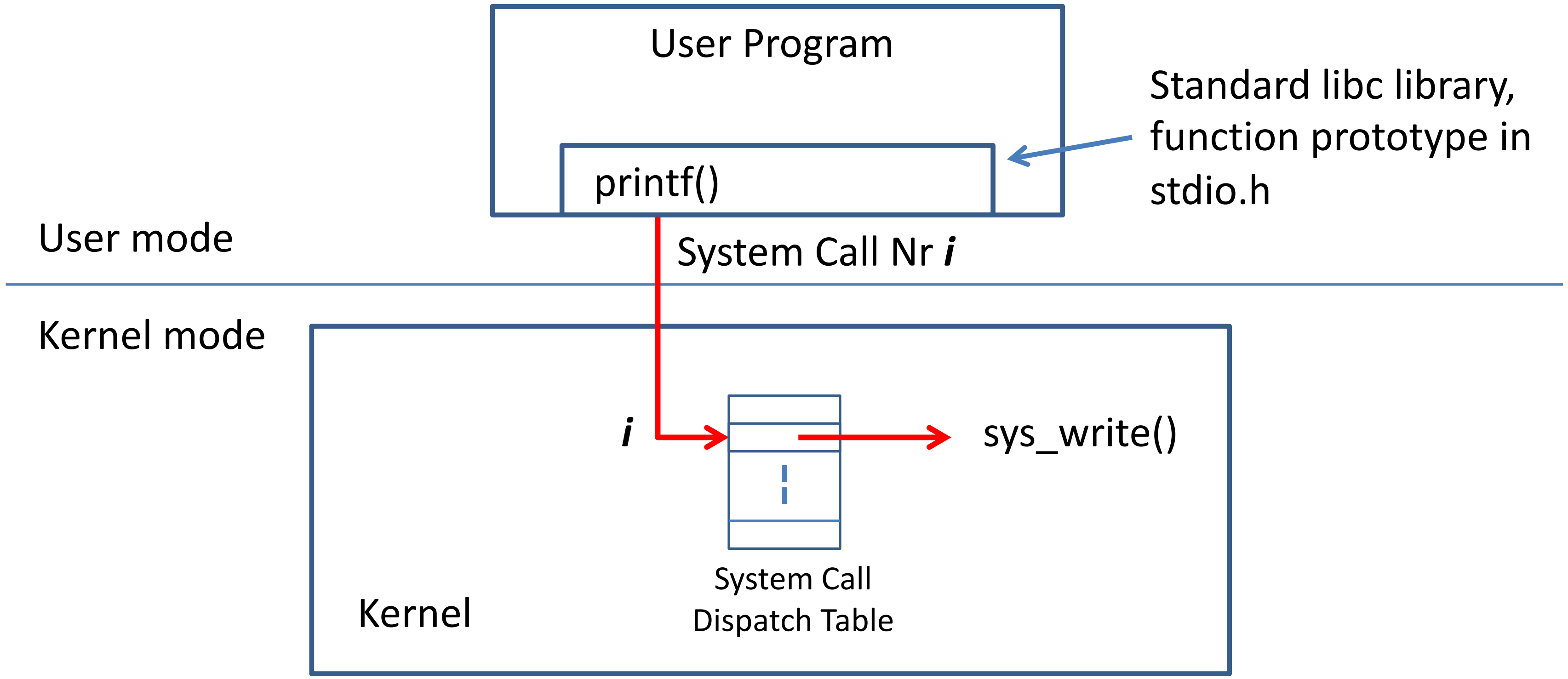
- Software interrupt due to system calls
  - Mode switch of processor hardware: System calls only allowed to execute in kernel mode
  - Mode bit managed be the processor
    - Provides the ability to distinguish between user and kernel mode
    - Privileged instructions only in kernel mode

Invoking a System Call

- Typically, a number is associated with each system call
  - The process invoking the system call must pass the *system call number* to the kernel to identify the corresponding system call service routine
- Operating system maintains a table of pointers to system call service routines, system call number is index for this table
- Operating system handles the invocation of the service routine and any return status / values

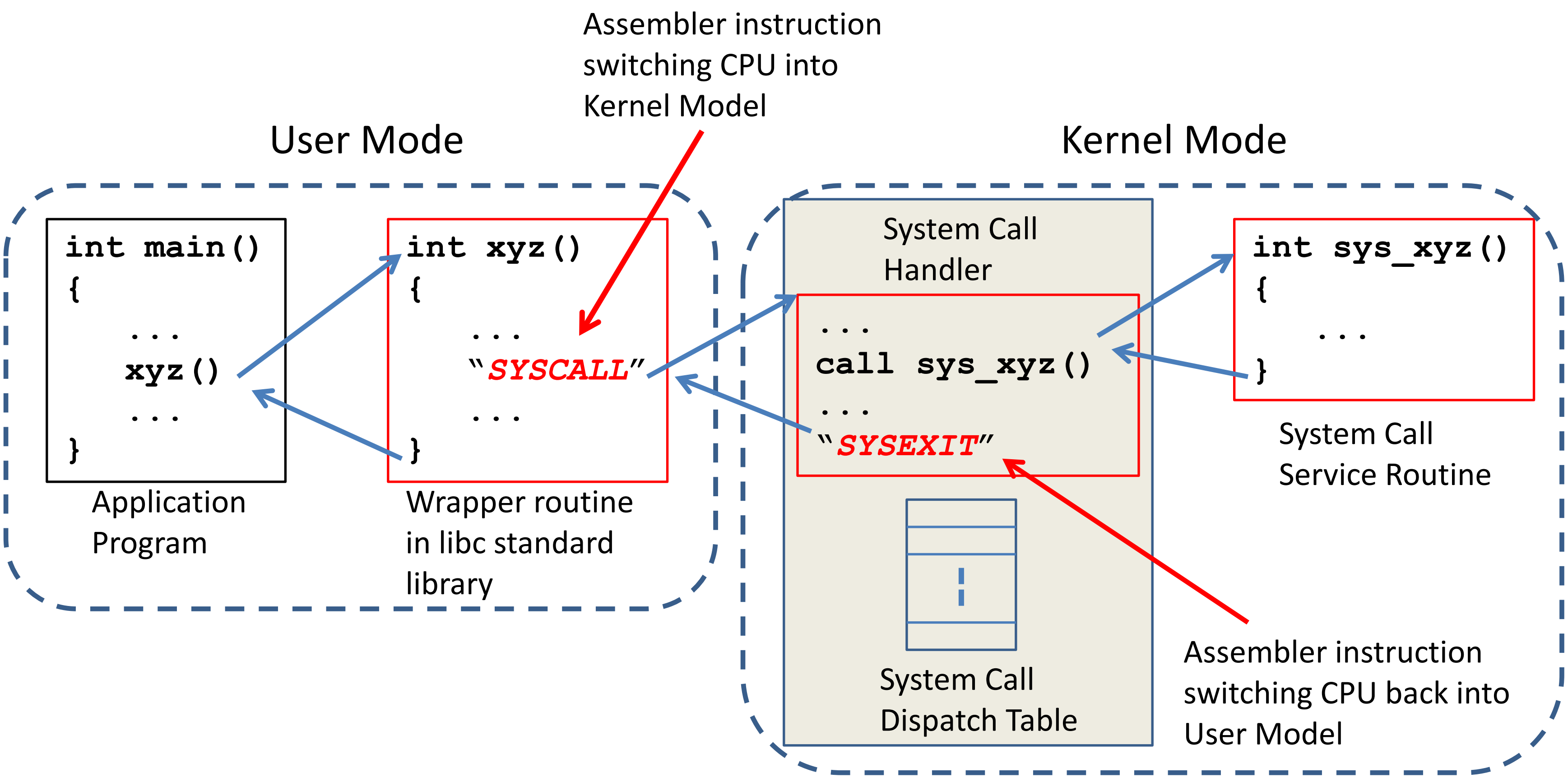


System Call Handling



- When a process running in user mode invokes a system call, the CPU switches to kernel mode and starts the execution of a kernel function

Invoking a System Call in Linux



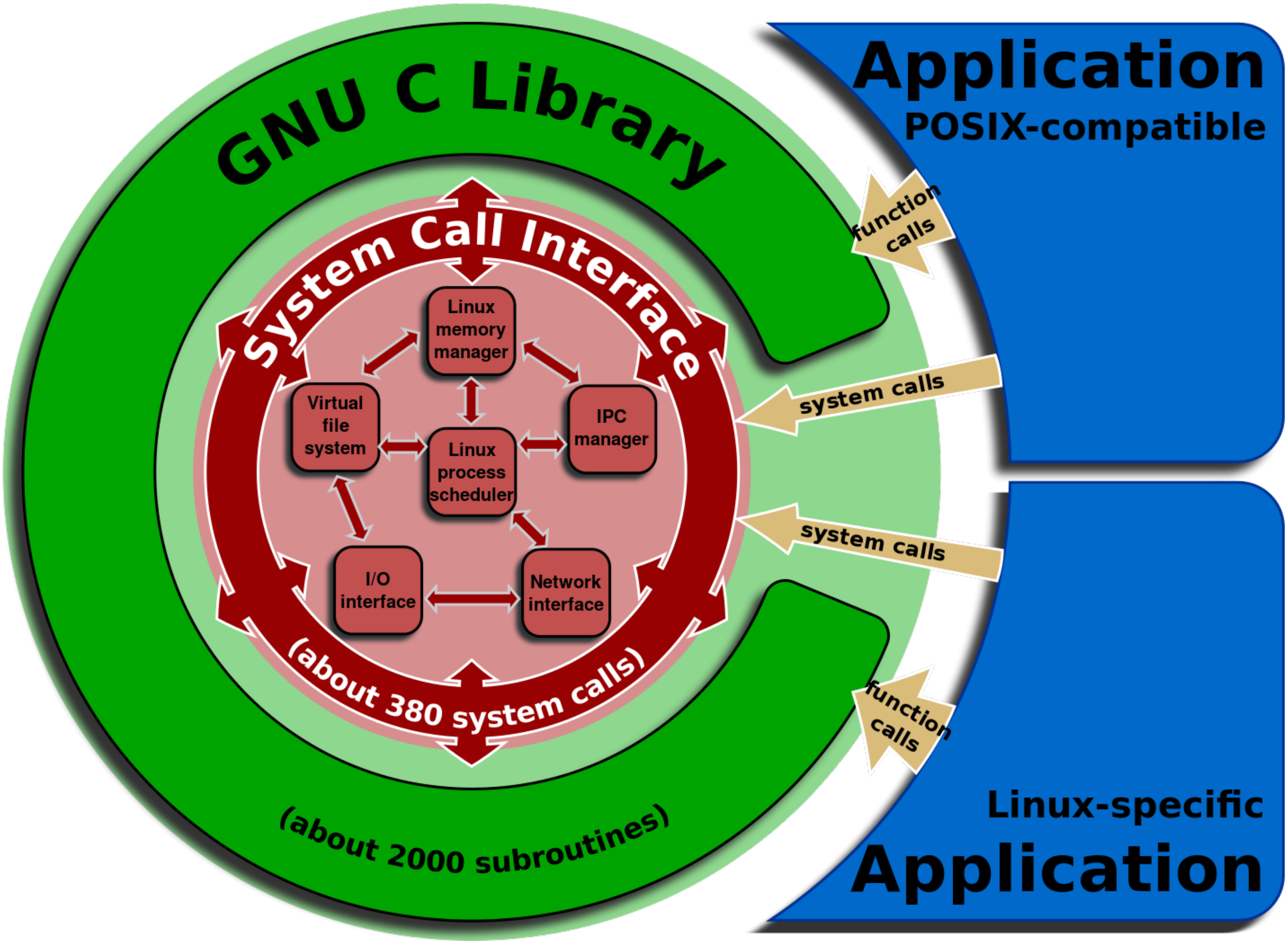
- Linux implements a system call handler to manage the invocation of system call service routines
- System Dispatch Table holds all the service routine addresses
  - System Call number is index into this table

Passing Parameters

- Three general methods:
  - Pass via CPU registers
  - Use a memory block:
    - Store parameters in memory in a table or memory block
    - Pass address of this memory block via CPU register to service routine
    - This approach is taken by Linux and Solaris
  - Use a stack
    - User program pushes parameters onto stack
    - System service routine pops parameters from stack

API’s and System Calls

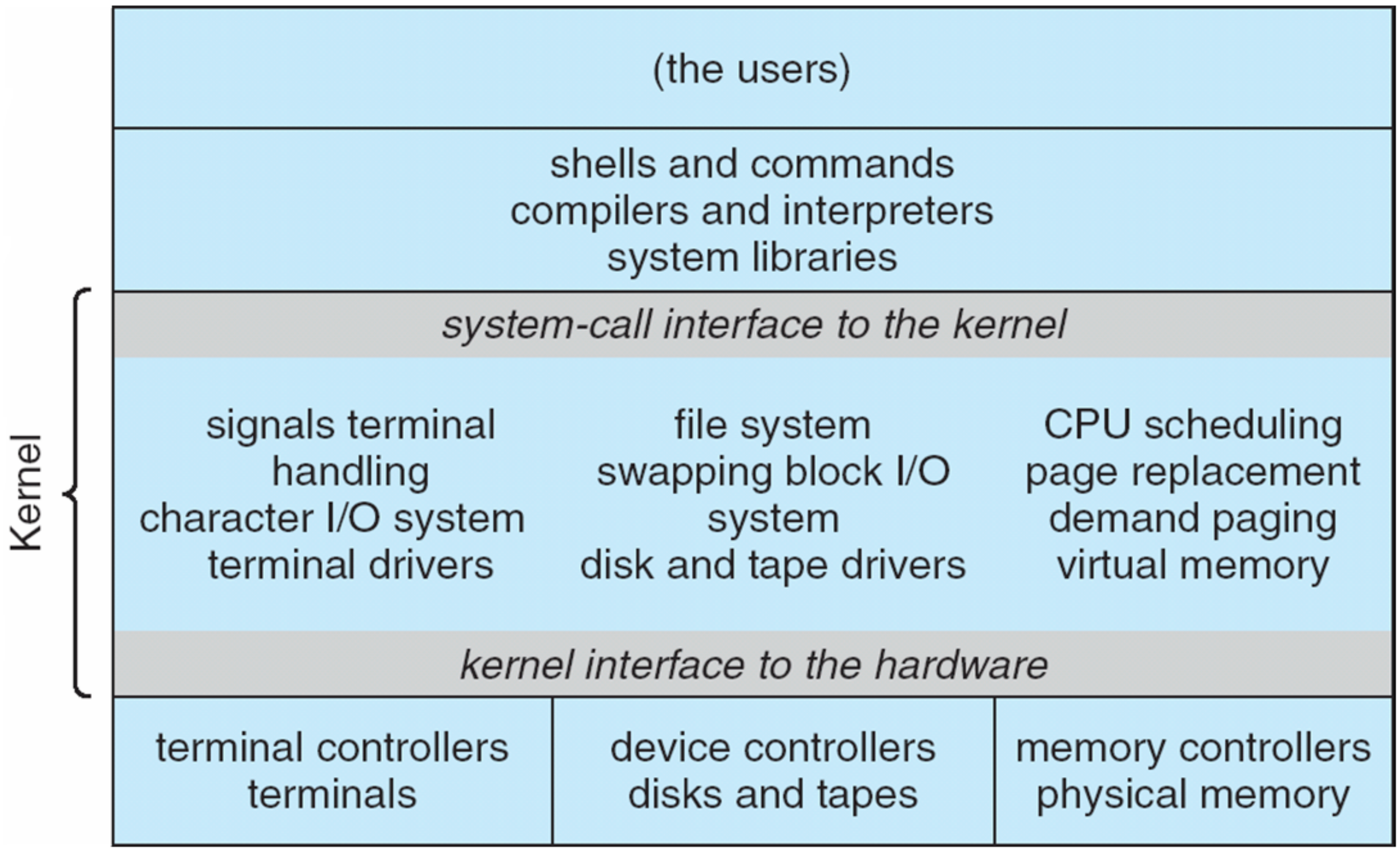
- Operating systems usually come with a library that implements an API of functions wrapping these system calls:
  - Typically written in a high-level language (C or C++)
    - Standard C Library
    - Unix / Linux: libc or glibc
  - Usually, each system call has a corresponding wrapper routine, which an application programmer can use in their programs
    - E.g.: printf()
- POSIX is a standard API implemented by many kernel architectures:
  - Many Unix kernels, Linux, Mac OSX, Windows NT
- Win32 is another important API



GNU C Library
(about 2000 subroutines)
System Call Interface
(about 380 system calls)
Linux memory manager
Virtual file system
IPC manager
Linux process scheduler
I/O interface
Network interface
Application
POSIX-compatible
function calls
system calls
system calls
function calls
Linux-specific
Application

Kernel Architectures

Unix Kernel



(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory
Kernel

Kernel Architectures

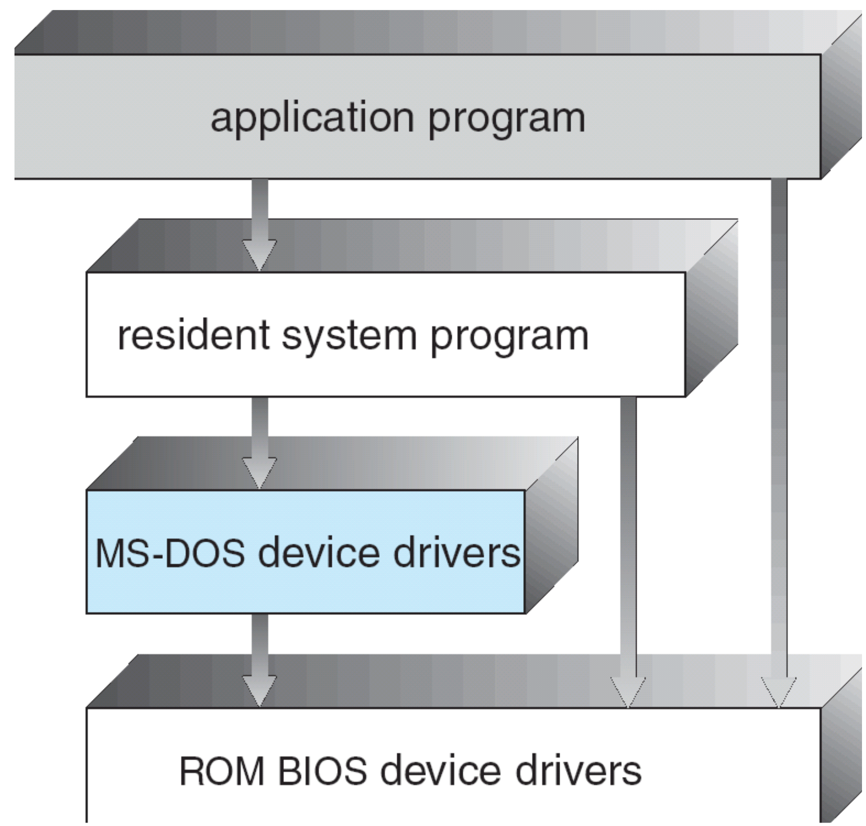
- Kernel the core element of operating system
- Various design and implementation approaches
  - Monolithic kernels
  - Layered approach
  - Microkernel
  - Kernel modules

Monolithic Kernel

- Most operating systems, until recently, featured a large monolithic kernel (most Unix systems, Linux)
- Provide
  - Scheduling
  - File system management
  - Networking
  - Device drivers
  - Memory management
  - Etc.
- Implemented as a single process
  - All functional components share same address space
- Benefit
  - Performance
- Problem
  - Vulnerability to failure in components

Simple Monolithic Structure

- Early operating systems were monolithic
- No well defined structure
- No layering, not divided into modules
- Started as small and simple systems
- Example: MS-DOS
  - Developed to provide most functionality in the least space
  - Levels not well separated, programs can directly access I/O devices



Introducing Layers

- Simple un-organised structures became infeasible
- Introduction of a layered approach
- Operating system is divided into a number of layers (levels), each built on top of lower layers
  - The bottom layer (layer 0) is the hardware
  - The highest layer (layer N) is the user interface
- Each layer uses only functions and services provided by a lower layer
- All or most of the layers operate in kernel mode
- Examples
  - MULTICS, VAX/VMS

Layered Approach

- Approach used by original Unix kernel
  - Minimal layering, thick monolithic layers, no clear separation – circular dependencies, difficult to debug and extend
- Better approach - strict layering
- Difficulty
  - How to define layers appropriately?
  - Layering is only possible if there is a strict calling hierarchy among system calls and no circular dependencies
- Example
  - The TCP/IP networking stack is a strictly layered architecture

Layered Approach - Problems

- Circular dependencies
  - Example disk device driver
    - Device driver may have to wait for I/O completion, invokes the CPU scheduling layer
    - CPU may need to call the device driver to swap processes in and out to hard disk
- The more layers the more indirections from function to function and the bigger the overhead in function calls
- Backlash against strict layering: return to fewer layers with more functionality

Microkernel

- A microkernel is a reduced operating system core that contains only essential OS functions
- Idea: minimise kernel by executing as much functionality as possible in user mode
  - Run them as conventional user processes
  - Processes interact only via message passing (IPC)
- Many services are now external processes
  - Device drivers
  - File systems
  - Virtual memory manager
  - Windowing systems
  - Security services etc.
- Example: Mach operating system

Mach Kernel

- Developed at Carnegie Mellon University 1985
- Research kernel
- Various versions of it were developed further
  - Microkernel as well as non-microkernel versions
- Notably
  - NeXTSTEP / Mac OSX, FreeBSD (not a microkernel, but provides microkernel IPC to applications)
- Problem of Mach kernel: IPC (Inter-Process Communication) overhead

Microkernel System Structure

- Operating system components external to the microkernel are implemented as server processes
  - These processes interact via message passing (IPC)
- Microkernel facilitates the message exchange
  - Validates messages
  - Passes messages between components
  - Checks whether message passing is permitted
- Grants access to hardware
- Microkernel effectively implements a client-server infrastructure on a single computer

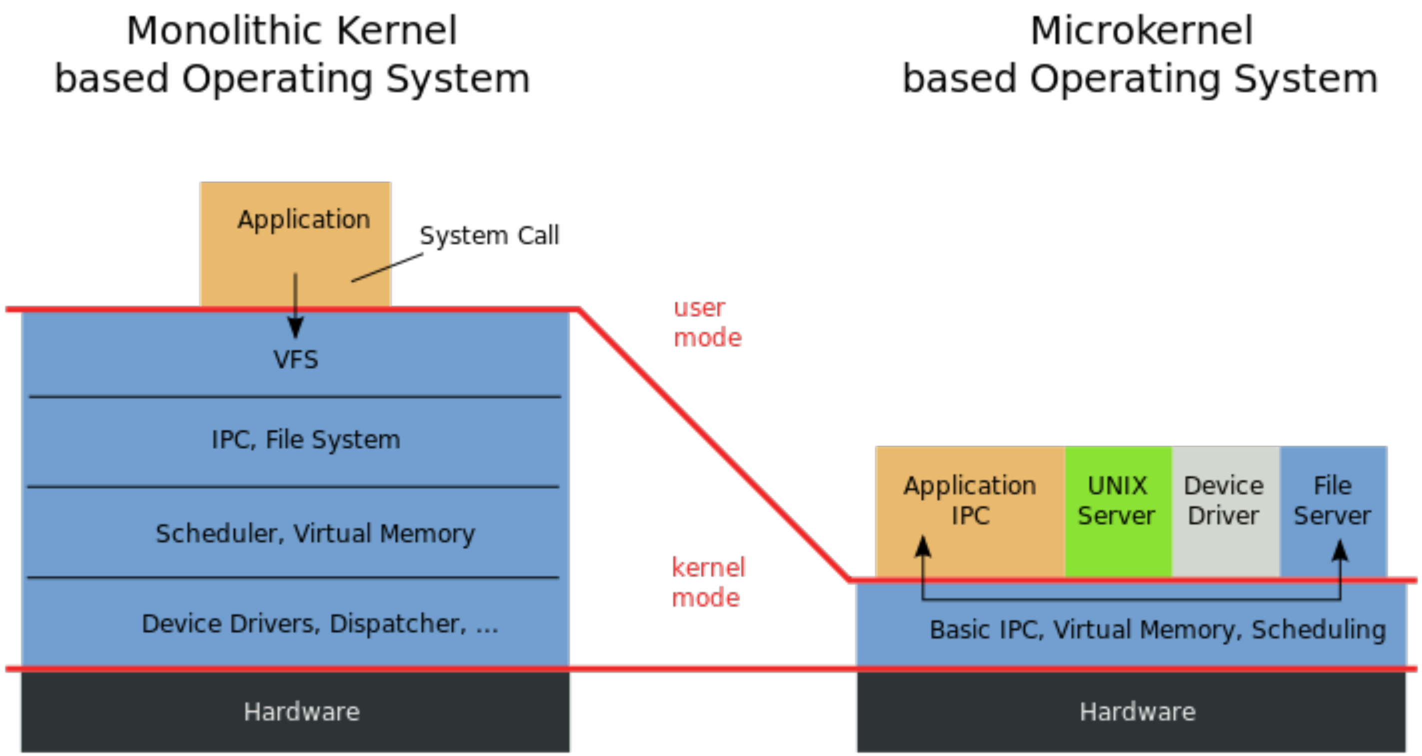
Microkernel

- Benefits
  - Uniform interfaces
    - Processes pass messages, no distinction between user-mode and kernel-mode services, all services are provided via message passing as in a client-server infrastructure
  - Extendibility
    - Easier to extend, new services introduced as new applications
  - Portability
    - Only the microkernel has to be adapted to a new hardware
  - Reliability and security
    - much less code runs in kernel mode, program failures occurring in user mode execution does not affect the rest of the system

Microkernel Design

- Minimal functionality that has to be included into a microkernel
  - Low-level memory management
    - Mapping of memory pages to physical memory locations
    - All other mechanisms of memory management are provided by services running in user mode
      - Address space protection
      - Page replacement algorithms
      - Virtual memory management
  - Interprocess communication (IPC)
  - I/O and interrupt management

System Call Monolithic vs Micro Kernel



Microkernel

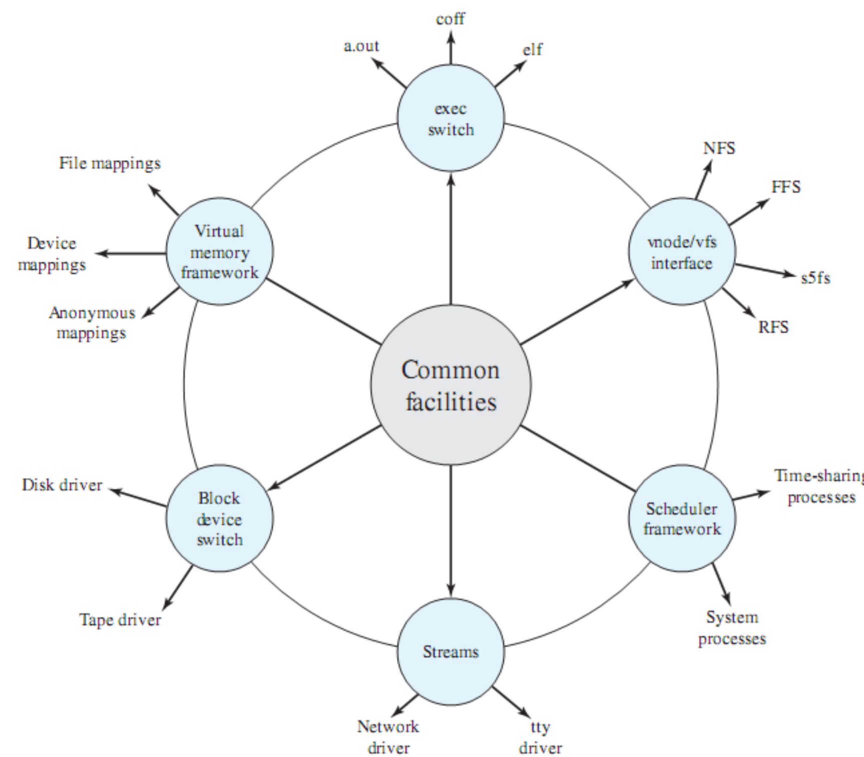
- Problems
  - Performance overhead of communication between system services
    - Each interaction involves the kernel and a user mode / kernel mode switch
    - System services running in user mode are processes, operating system has to switch between them
  - Solution: reintegration of services running in user mode back into the kernel
    - Improves performance: less mode switches, services integrated in kernel share one address space (one process)
    - This was done with the Mach kernel
  - Solution: make kernel even smaller – experimental kernel architectures (Nano kernels, pico kernels)

Modular Kernel Design

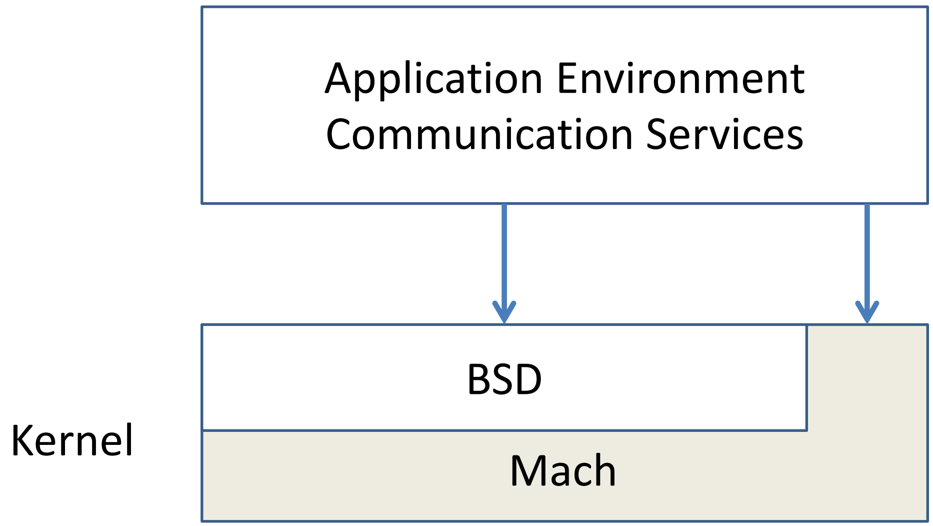
- Many operating systems implement kernel modules
  - E.g.: Linux
    - Each core component is separate
    - Communication via defined interfaces
    - Loadable on demand
- Modules are somehow a hybrid between the layered and microkernel approach
  - Clean software engineering approach
  - But: modules are inside the kernel space, they don't require the overhead of message passing
  - Compromise with performance benefits

Modern Unix Kernel

- Modern Unix kernels have a modular architecture
- Common facilities as the inner core of the kernel
- Rest of system services added as modules
- See Linux



Modular Approach

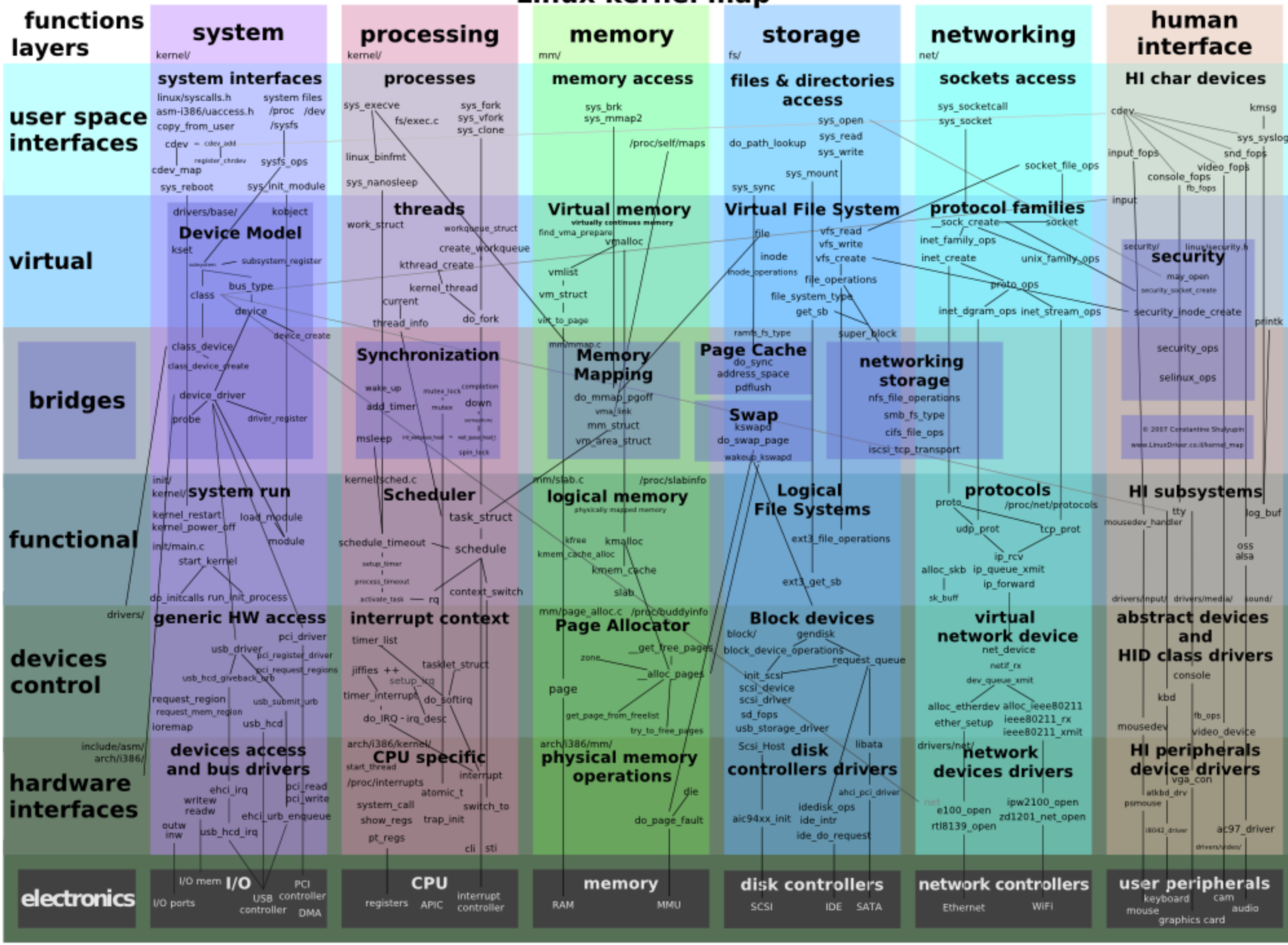


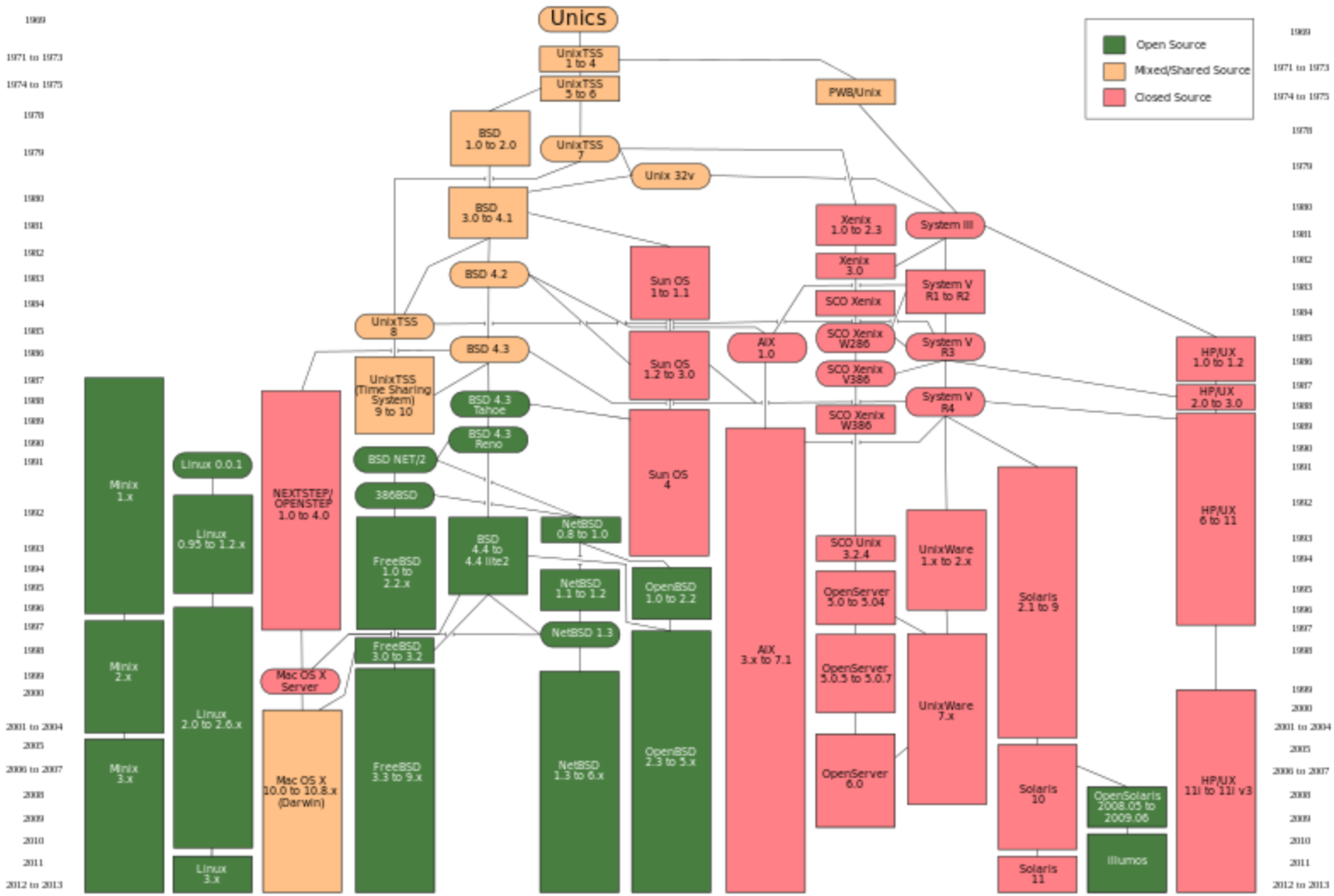
- Mac OSX
  - takes a hybrid approach, has a Mach kernel (microkernel) combined with a BSD interface
  - BSD (**Berkeley Software Distribution**, sometimes called **Berkeley Unix**): provides support for command line interface, networking, file system, file system, POSIX API and threads
  - Mach: memory management, Remote procedure Call (RPC), Interprocess communication (IPC), message passing

OS Functionality

- Linux Kernel Map
  - http://www.makelinux.net/kernel_map/
- Also available at:
  - http://upload.wikimedia.org/wikipedia/commons/5/5b/Linux_kernel_map.png
  - http://i.imgur.com/4sftcoo.jpg

Linux kernel map





http://en.wikipedia.org/wiki/File:Unix_history-simple.svg