P1L2 Intro to OS

Goal

- What is OS?
- Key comonents of OS
- Design & Implementation considerations of OS

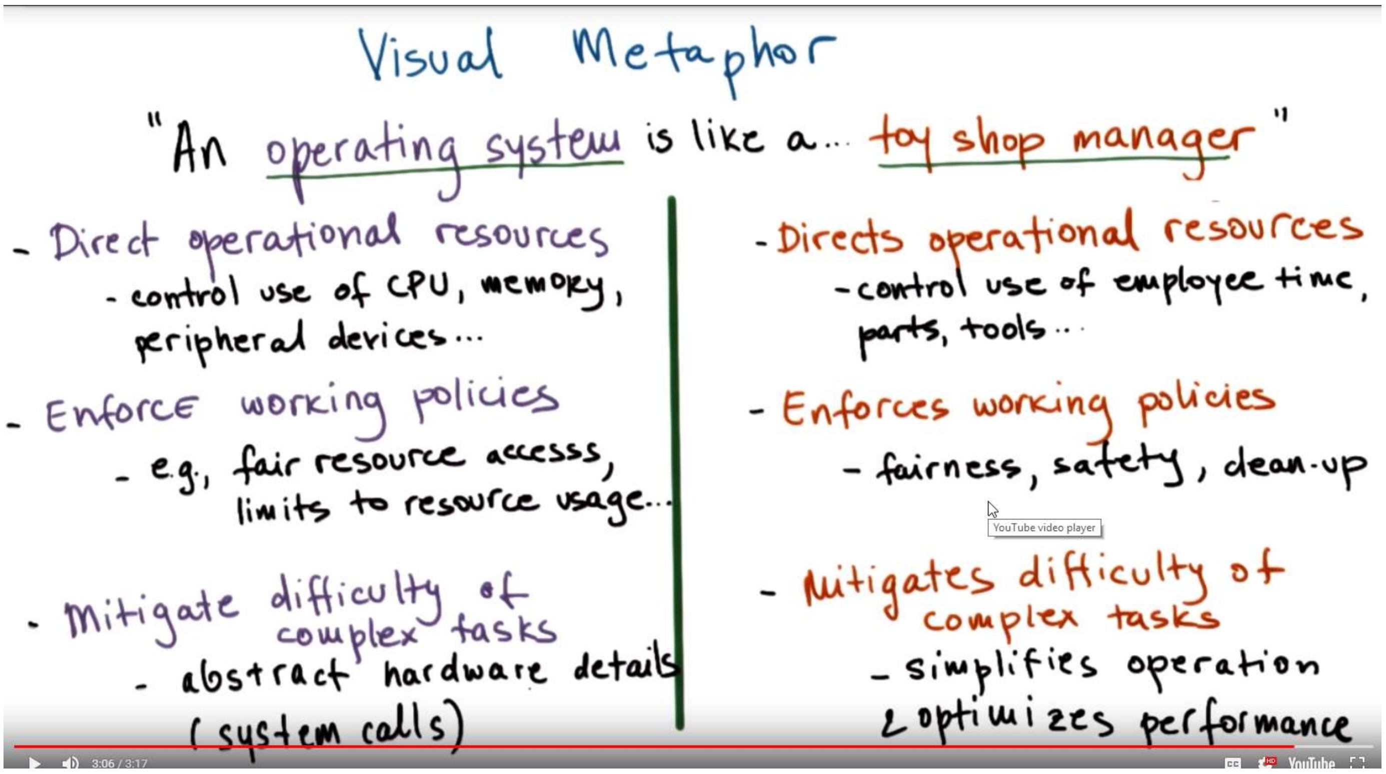
1. What is OS?

1.1 Simple OS Definition

Software that:

- abstract -----simplify what the hardware actually looks like
- arbitrates ----- manage, oversee and control hardware use.
  the underlying hardware system.

1.2 Toy shop manager metaphore



Visual Metaphor

"An operating system is like a... toy shop manager"

- Direct operational resources
  - control use of CPU, memory, peripheral devices...
- Enforce working policies
  - e.g., fair resource accesss, limits to resource usage...
- Mitigate difficulty of complex tasks
  - abstract hardware details (system calls)

- Directs operational resources
  - control use of employee time, parts, tools...
- Enforces working policies
  - fairness, safety, clean-up
- Mitigates difficulty of complex tasks
  - simplifies operation & optimizes performance

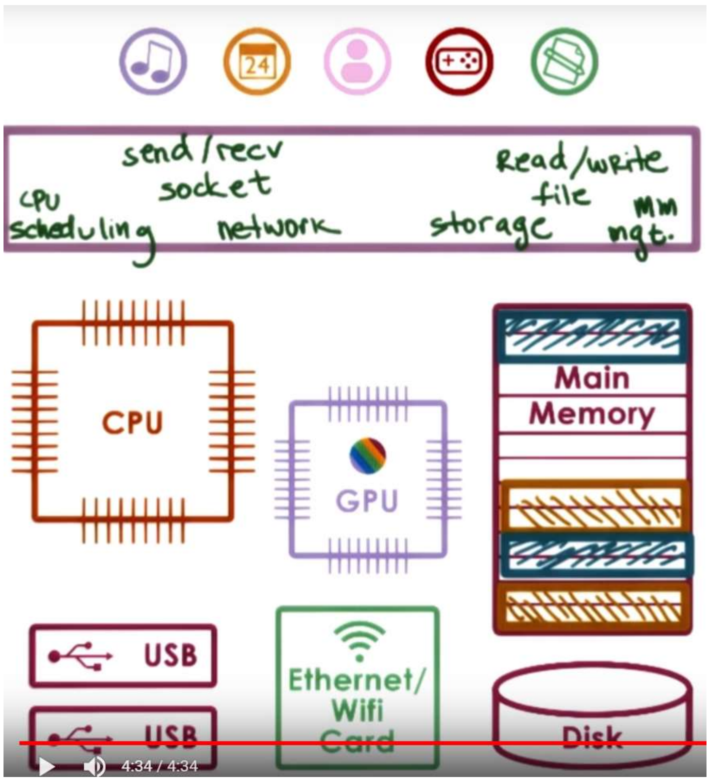
1.3 What is Operating System

> ***In one line:*** Layer between applications and hardware components.

What is does?

- **Hide Hardware complexity**

    - a bit like packaging mundane codes when you write a program.
- **Resource Management**
    - Here are some hardware components



    - OS manages hardware on behalf of one or more applications
    - Follows defined policies
- **Provide isolation and protection**
    - so that one application doesn't access another application's memory space and do something crazy.

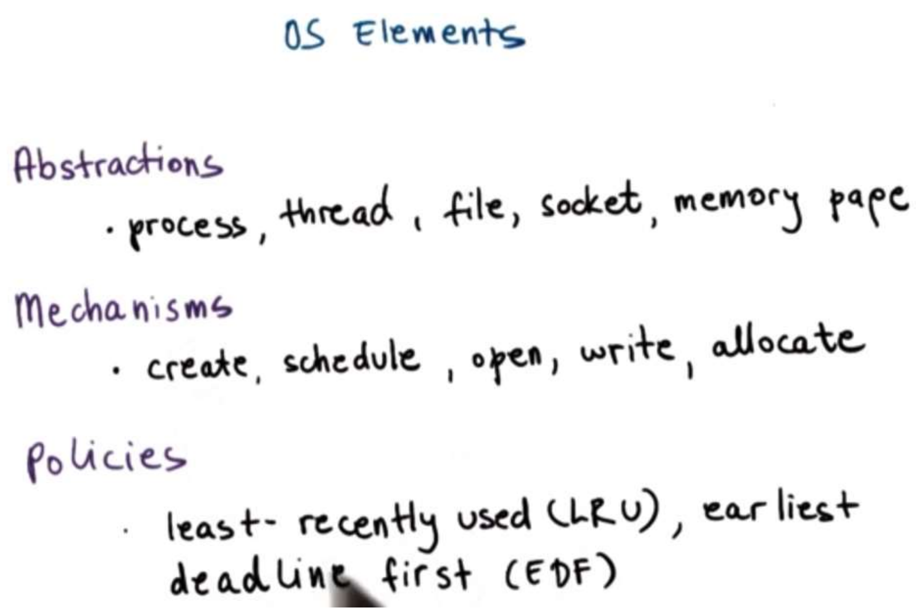
> **Human Language**: OS is like the President of a computer system, it controls and manages the common resources that all its people (the applications) needs. He might look like a tyrant because he hides all the valuable resources from his people, but we all know it's for everybody's good. Without the beloved President, people will fight over resources (hardware components) and drive each other nuts (think about what happens if someone breaks into your head and trash your memory).

1.4 Example: Types of OS

- Desktop
    - Microsoft Windows
    - Unix-based
        - mac OS X (BSD: Berkley System Distribution of Unix)
        - **Linux (different versions)**
- Embedded
    - Android
    - IOS
    - Symbian (RIP...)

---

2. OS Elements


OS Elements

Abstractions
- process, thread, file, socket, memory pape

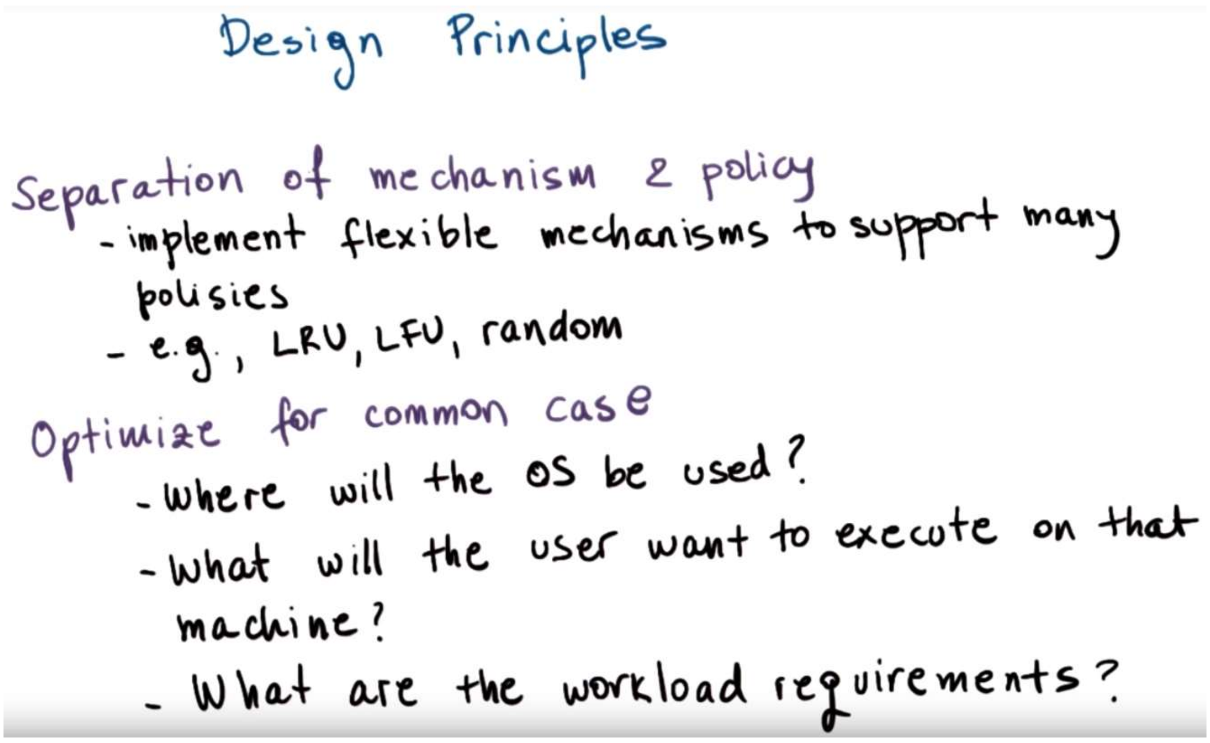
Mechanisms
- create, schedule, open, write, allocate

Policies
- least-recently used (LRU), earliest deadline first (EDF)

Think of abstractions as things/resources/nouns.
Mechanisms are verbs and operations the OS can do upon abstractions.
Policies are rules/conventions of OS operations on abstraction of hardware resources.

---

3. OS Design Principles


Design Principles

Separation of mechanism & policy
- implement flexible mechanisms to support many polisies
- e.g., LRU, LFU, random

Optimize for common case
- where will the OS be used?
- what will the user want to execute on that machine?
- What are the workload requirements?

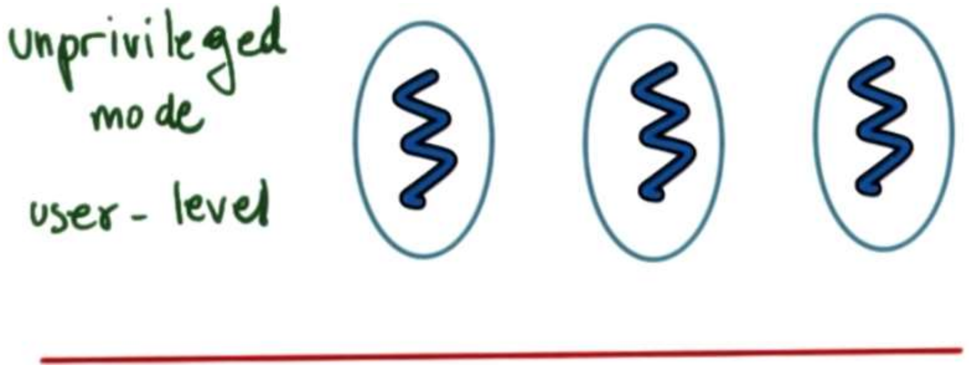
---

4. OS Protection Boundary & Interaction

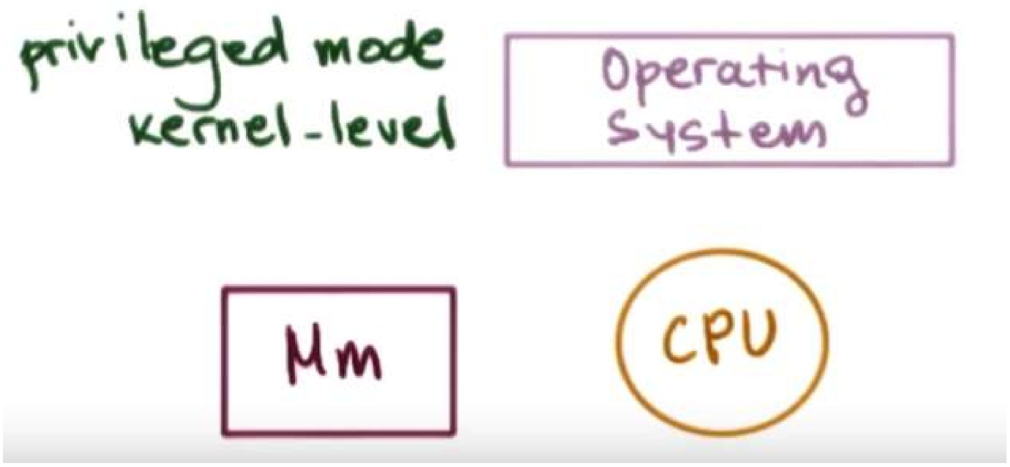
4.1 Two Distinguished Modes

- **User Mode:** unprivileged

    - Applications



- **Kernel Mode:** privileged
    - OS kernel
    - direct hardware access



4.2 Crossing/Interaction between two modes

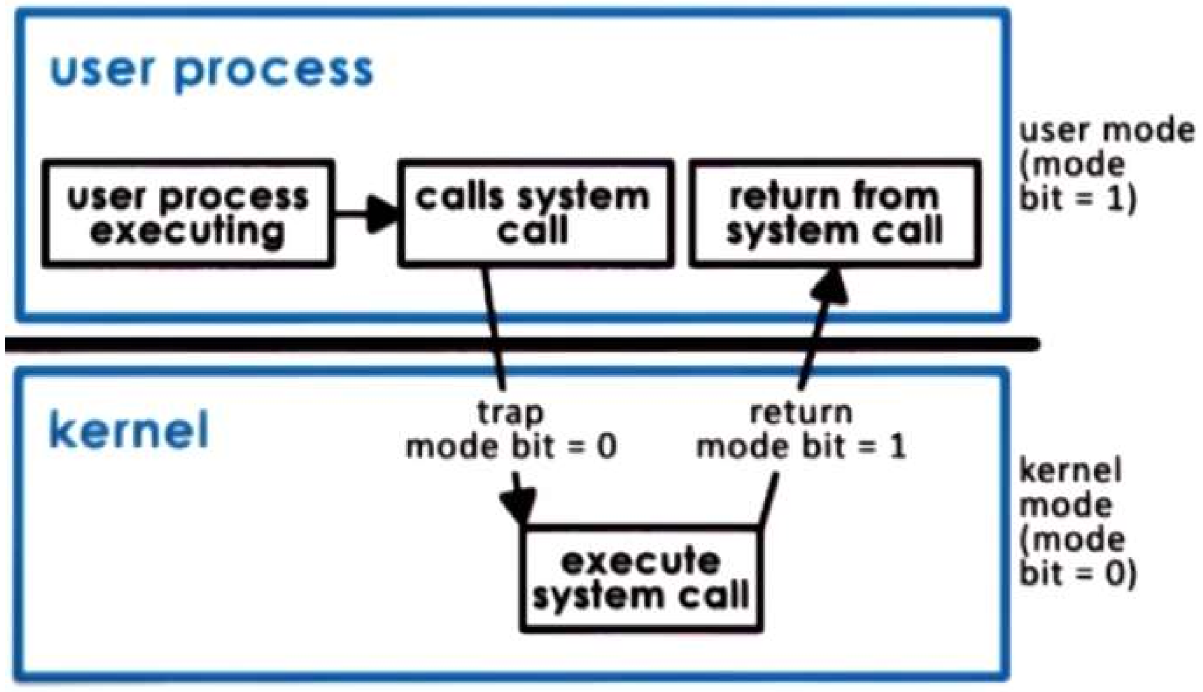
- **supported by hardware**
- **CPU privilege bit**
    - set in kernel mode – allow privileged operation
    - not set in user mode – forbidden from privileged operation
- **Trap:**
    - *Causation*: Attempt to perform privileged operation when in user mode
    - *What will happen?*
        - Application interrupted
        - Switch hardware control to operating system
    - *What next?*
        - OS check what cause the trap to occur
        - OS verify if it should grant that access or not
            - if deemed performing illegal operation, OS could terminate the process
- **System Call Interface** ---- Another Interaction between apps and OS
    - A set of operations that apps can explicitly invoke/call
    - Request the OS to perform certain service/certain privileged access on the application's behave.
    - Example: open, send, mmap
- **Signal**
    - Mechanism for OS to pass notifications into applications.
- **User/Kernel Transition Time/Instruction Overhead**

- It involves a number of instruction
- e.g. ~50-100 ns on a 2GHz machine running Linux
- System calls are not cheap!

- **Another cost of User/Kernel Mode Switching**
    - Cache Performance
        - Hot(fire): When application finds what it needs in the cache
        - Cold(ice): When application can't find what it nees in cache --> App have to retrieve data it needs from main memory which is a lot slower than cache.
    - Switching between Kernel/User modes will swap data/address currently in cache
    - It depends if the change is good or bad for the application's performance

4.3 System Call Flow Chart

4.3.1 What happens when an application make a system call?

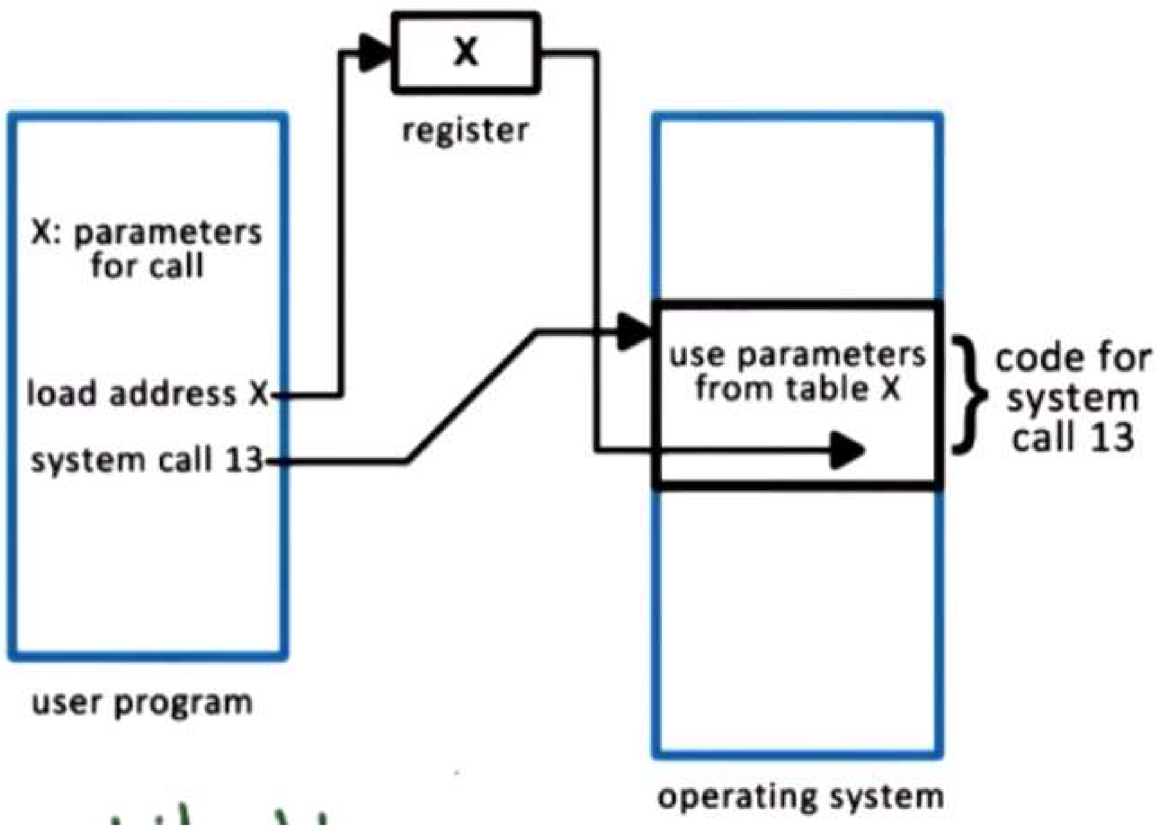


When an application make a system call:
control passed ot OS --> switch to privileged mode --> OS perform requested operation --> Result returned to application process --> application process continues in user mode.

The system call is not necessarily a cheap operation.

Synchronous Mode: The process that made the system call will wait until the system call to complete.

4.3.2 How is system calls carried out?

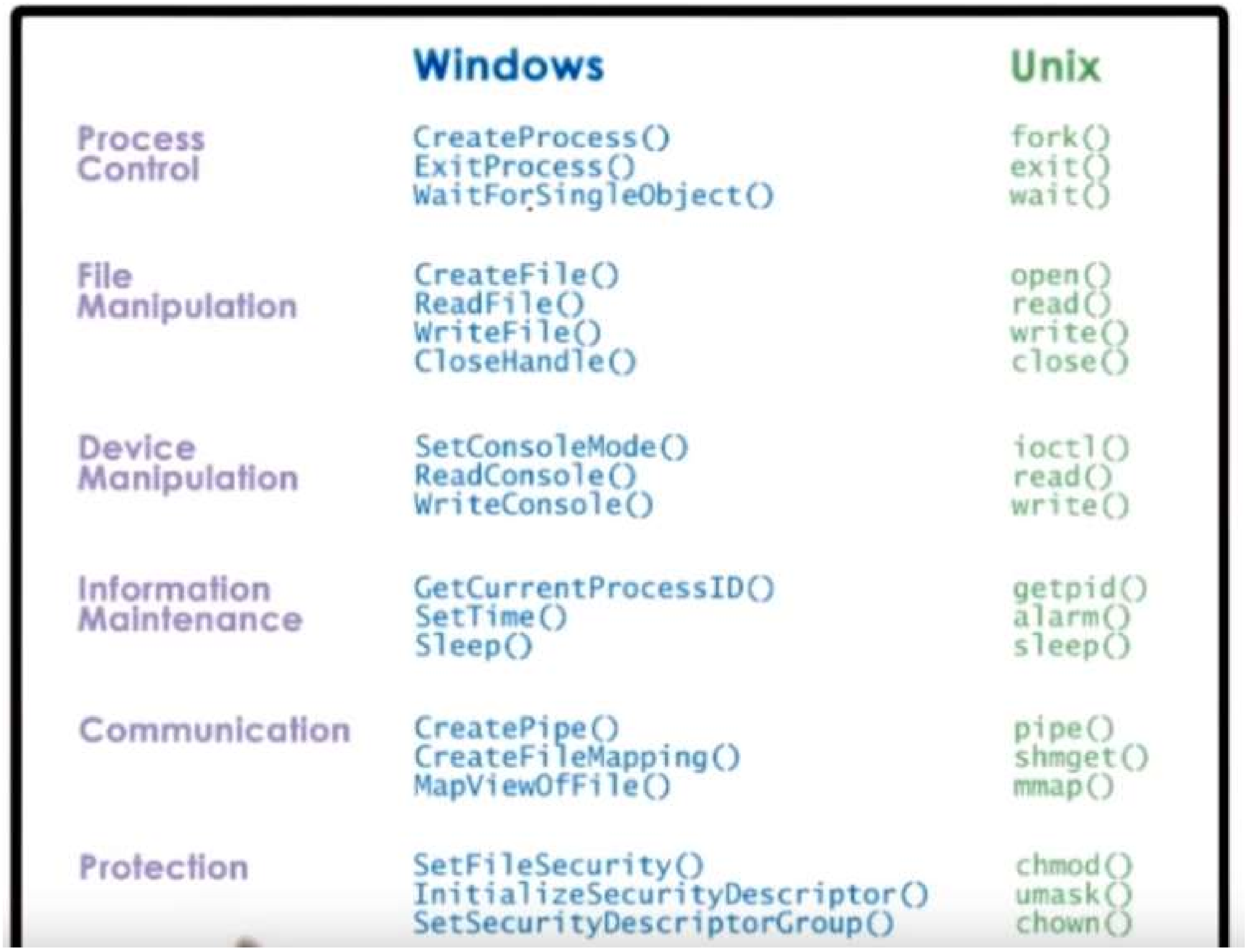


1. Application writes arguments
    - The arguments can either be passed directly between user program and OS
    - Or arguments can be passed indirectly by specifying their addresses
2. Application save relevant data at well-defined location
    - The well defined location is where the system call kernel gets the necessary system call arguments
3. Make the actual system call with system call number
    - The OS kernel finds out which system call is requested and how many arguments it needs based on the system call number

4.4 OS Services

- OS must provide apps with access to underlying hardware components.
- The access is provided with services via system calls.

Here are some sample system calls in Windows and Unix:



	Windows	Unix

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Information Maintenance	GetCurrentProcessID()
	SetTime()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

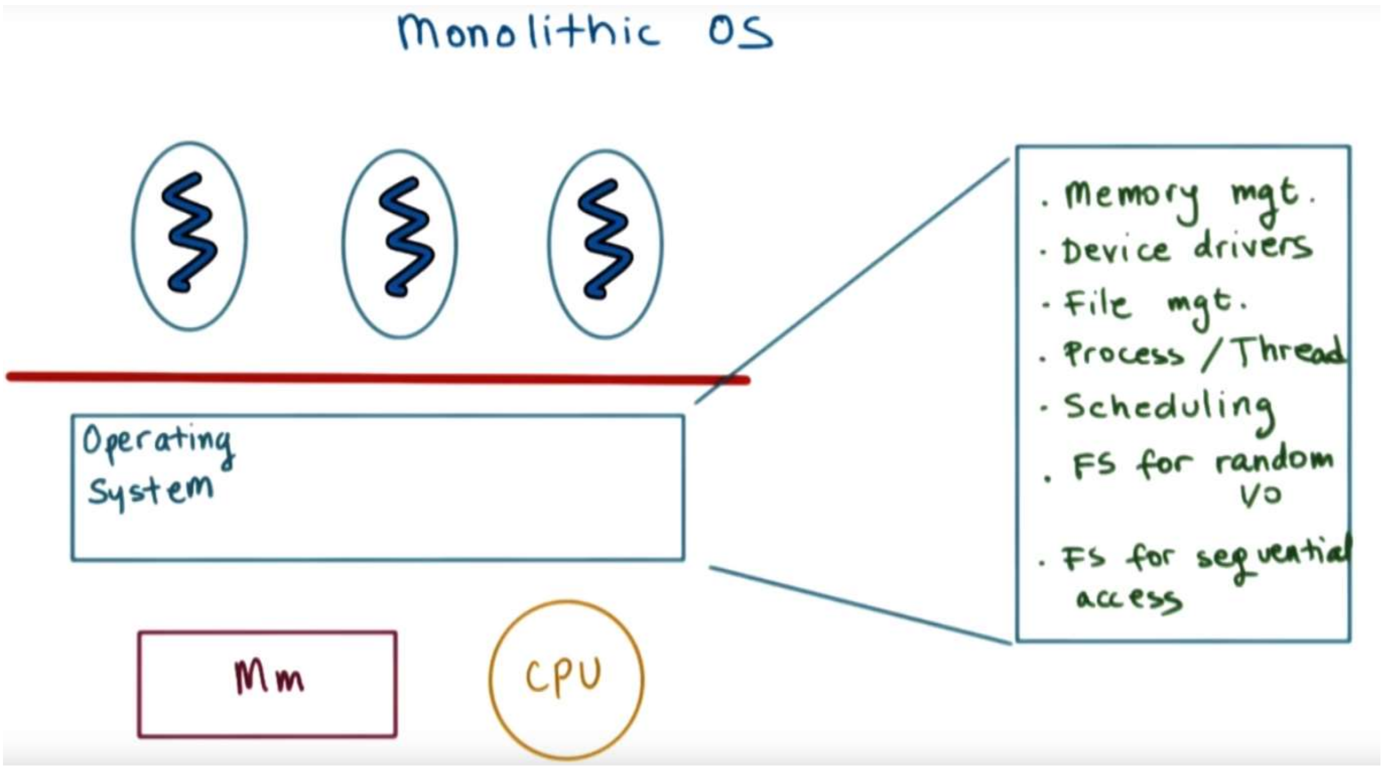
Protection	SetFileSecurity()
	InitializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

Note that the service provided in two different OS are very similar.

---

5 Different Types of OS Organizations

5.1 Monolithic OS



What is it?

Every service that the apps or the hardware will demand is already part of the OS.
e.g. Such a monolithic OS may include several possible file systems, like for sequenctial and random I/O.

The good

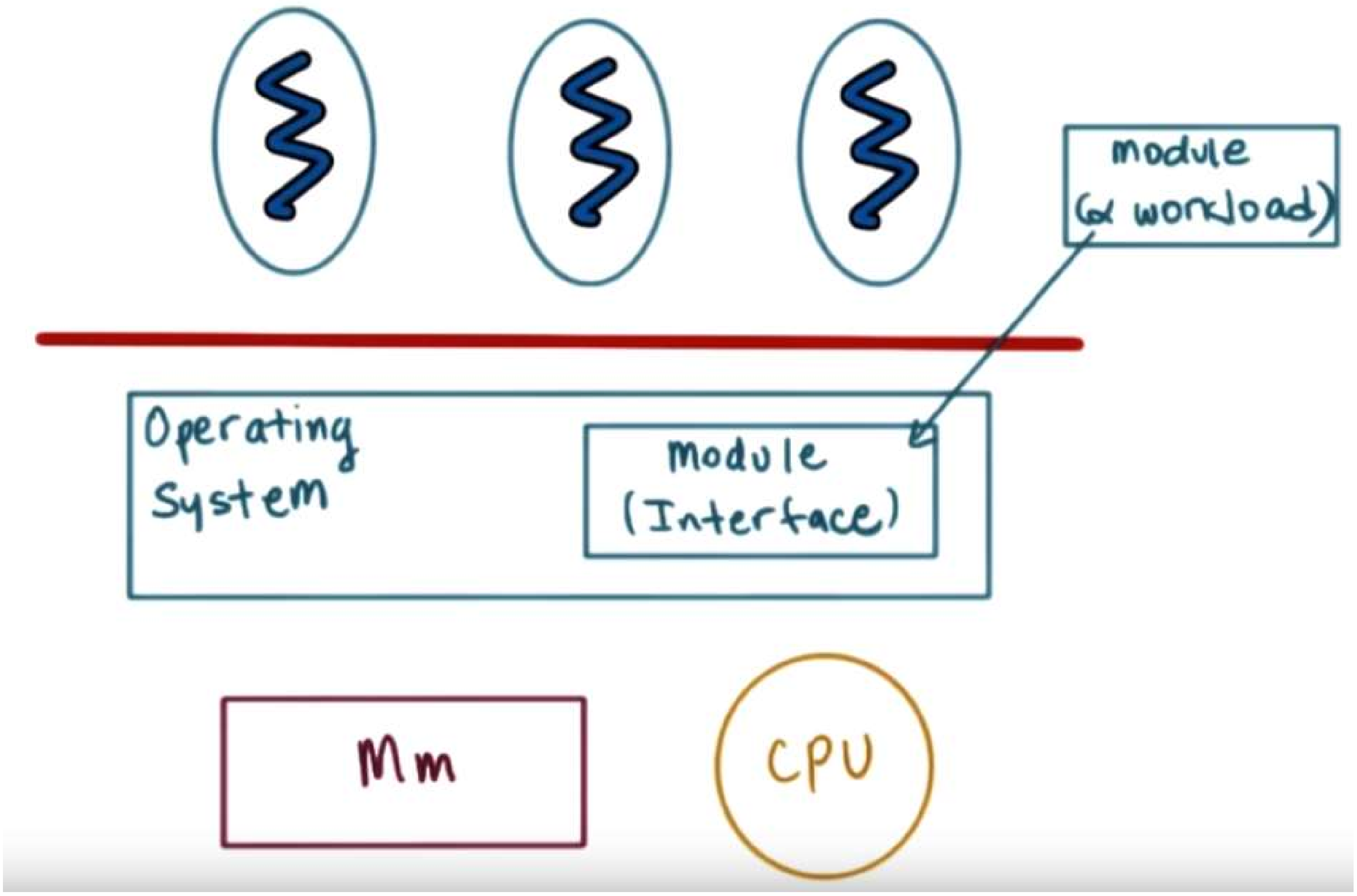
- Everything included
- Possibilities for inlining, compile-time optimizations

The bad

- The OS could be really really large
- Too much state, too much code
- Hard to maintain, customize, debug or upgrade
- Large memory requirements and hurts applications' performance

5.2 Modular OS

Representitive OS: Linux
It's more commonly used today.
Overall it delivers significant improvement over the monolithic approach.



What is it?

- It has some basic services and APIs already part of it.
- Everything can be added as a module: e.g. file systems that suits customized workload needs.
- The OS specifies certain interfaces that all module must implement to be part of the OS.
- We can install modules dynamically depending on the workload

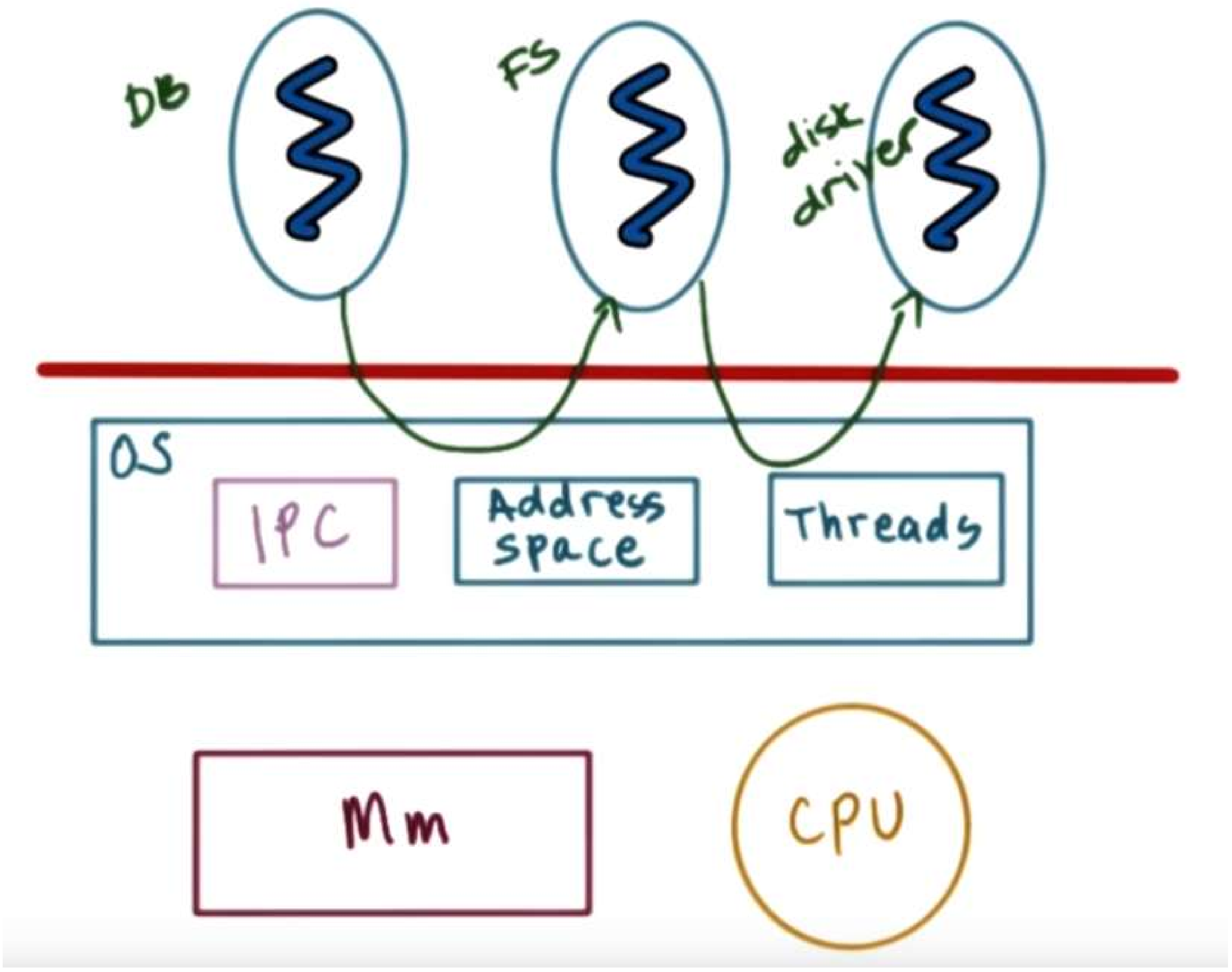
The good

- Easy to maintain and upgrade
- smaller code base
- less resource intensive —Leaving more resources for applications

The bad

- Level of interaction required by modulation can reduce some opportunities for optimizations.
  - This can ultimately have some impact on performance but not significant.
- Maintenance difficulty coming from disparate code bases.

5.3 Microkernel



What is it?

- It only require the most basic primitives at the OS level, such as executing applications, address space and context(thread).
- Everything else will run outside of OS kernel at user level. (e.g. DB, filesystem, devise drivers)
- It requires los of inter-process interatctions, so the OS supports inter-process communications as one of its core abstractions and mechanisms.

The good

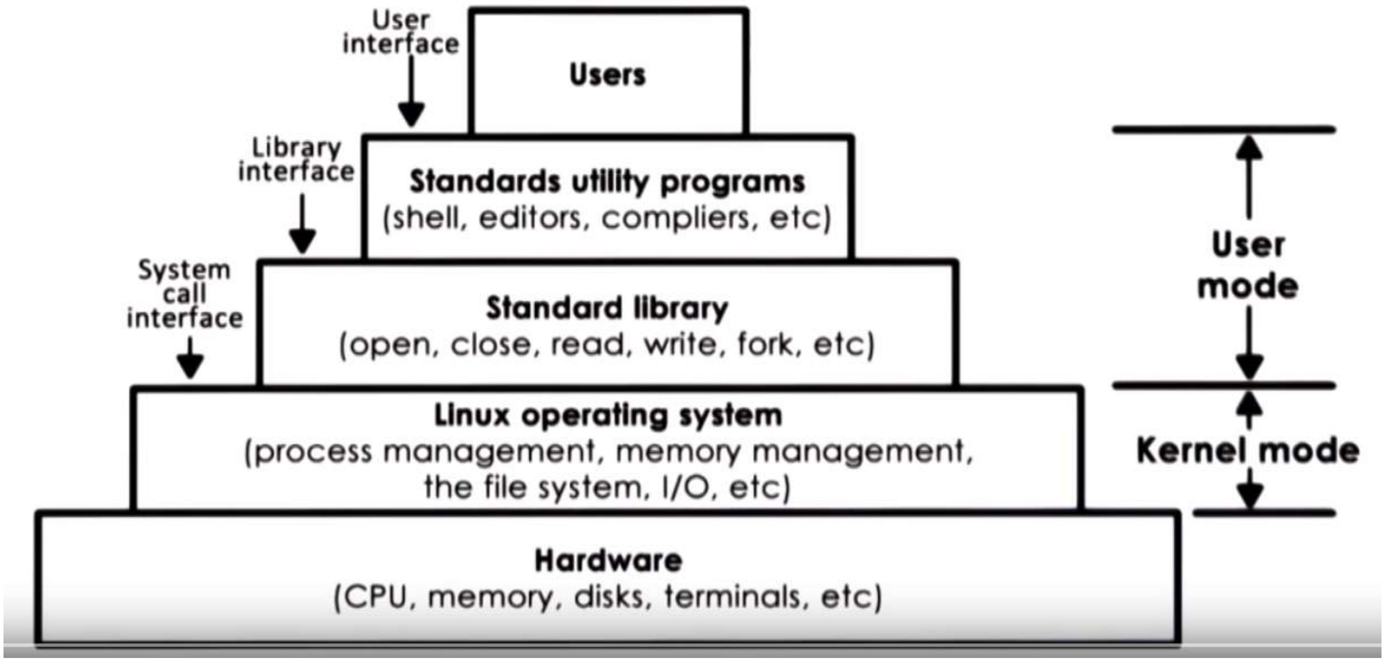
- Small size
  - lower overheads
  - better performance
  - easy to verify and test the code

The bad

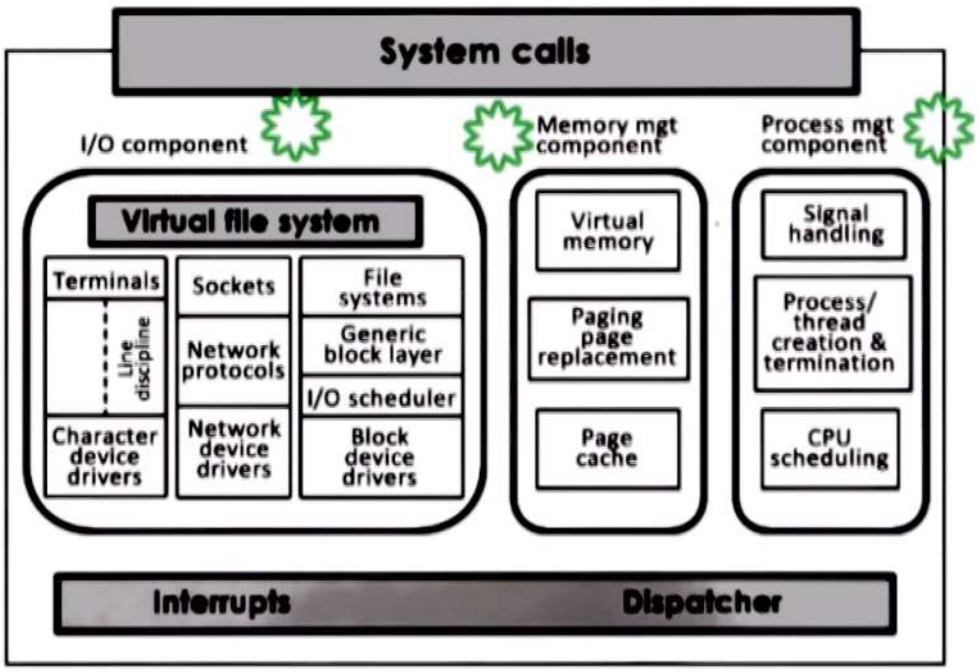
- Questionable portability —it's typically very specialized, customized to the underlying hardware.
- Software complexity — from multiple versions of micro-kernels for different platforms
- Frequent inter-process communication --> system calls to IPC services–> frequent user/kernel crossing–> they are not cheap.

Linux and Mac OS Architectures

Linux

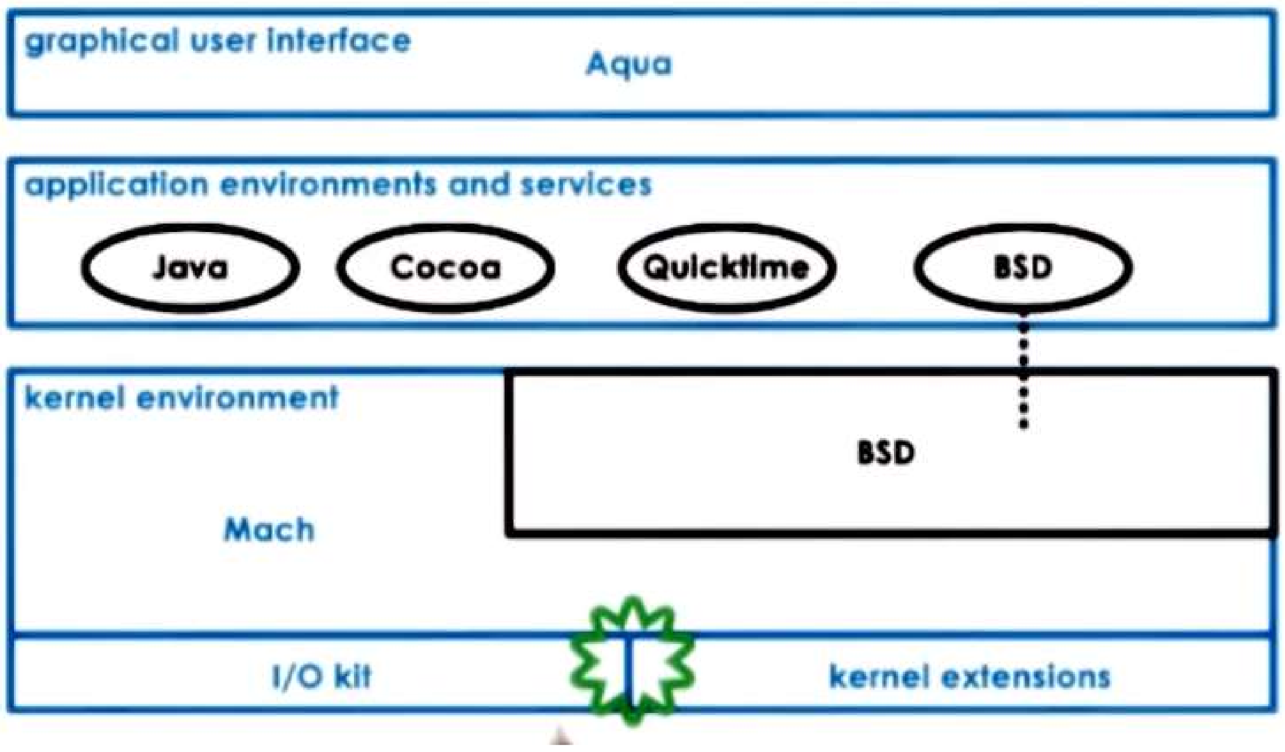

User interface
Users
Library interface
Standards utility programs (shell, editors, compliers, etc)
System call interface
Standard library (open, close, read, write, fork, etc)
Linux operating system (process management, memory management, the file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)
User mode
Kernel mode

Linux Kernel Components



Each of the components can be independently modified or replaced.

Mac OSX



- Mach: Mac micro module implements key primitives like memory management, thread scheduling and interprocess communication mechanisms including for what we call RPC.
- BSD: provide unix interoperability via a BSD commandline interface, POSIX API support as well as network I/O.
- I/O kit and kernel extensions: environments for development of drivers and kernel modules that can be dynamically loaded into the kernel.