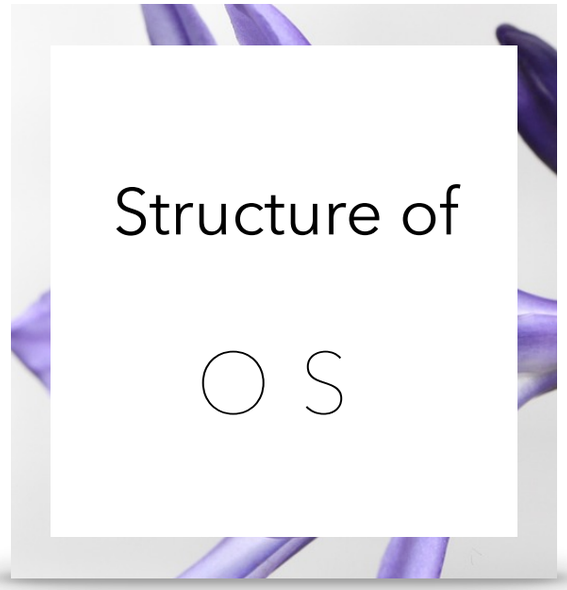


50.005 CSE

Natalie Agus
Information Systems Technology and Design
SUTD

OS SERVICES

What can the OS do?

helps users use the computer through its UI

load, run, exec, exit, error handling

create, del, rename, search, list, manage permission

Execute

Programs

I/O

Operations

File System

Operations and manipulation

Comms

Of processes via shared memory

note: highlighted tasks

+ some native I/O (write/read to disk are the super important part of kernel

they are the only ones implemented in microkernel, i.e: these services must be implemented in kernel mode (by the kernel) and cannot be compromised to be in user mode

Security & Error Detection

Handles debugging facilities

and recover from errors

Resource

Allocations

supports:

1. concurrency: program executions multiplexed
2. parallelism

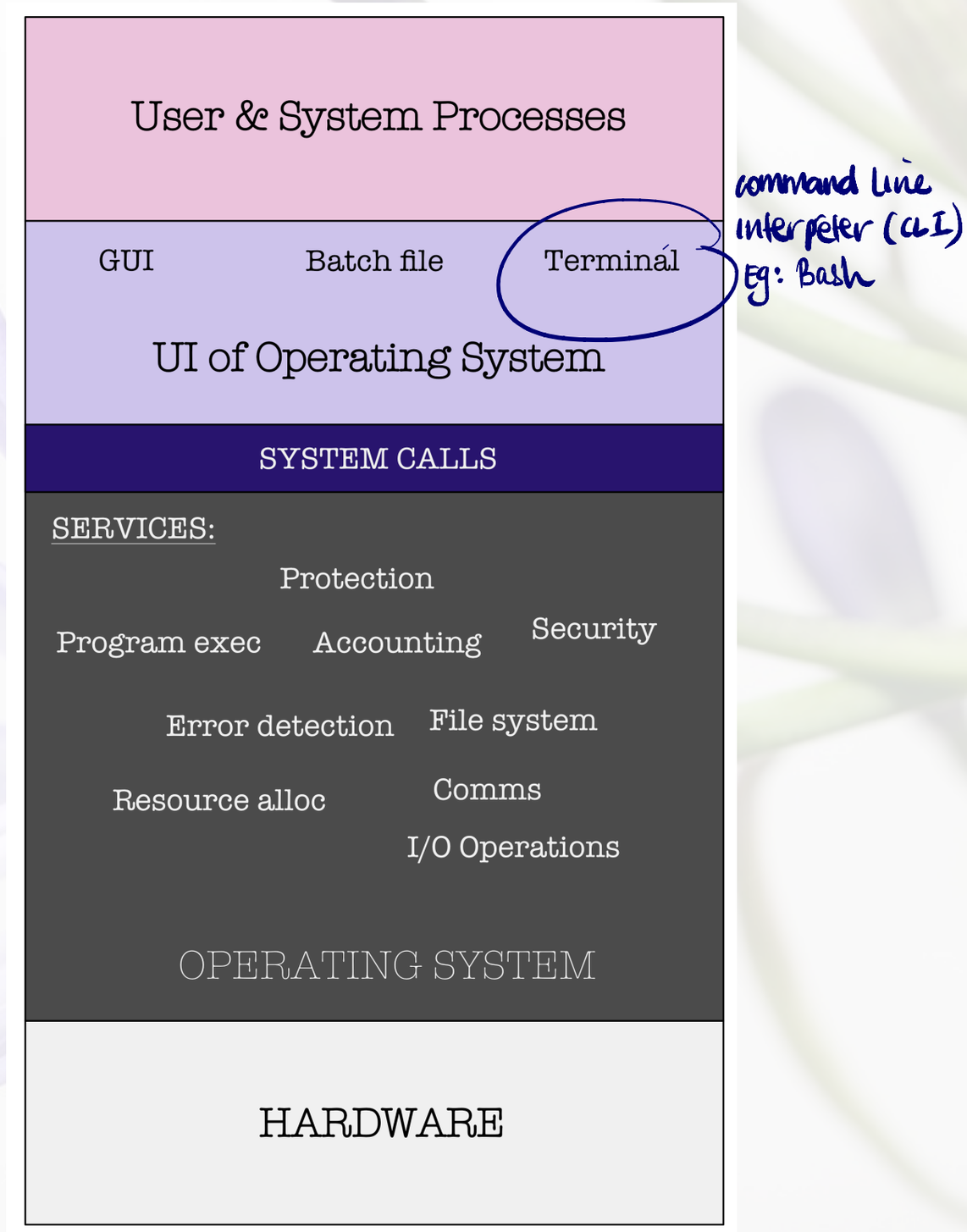
Accounting

Keeps track of processes

The OS acts as an intermediary between these two:

Hardware (regs, caches, RAM, I/O devices, CPU, etc)

User programs (browser, Spotify, Matlab, anything installed in your com)



command line interpeter (CLI)
Eg: Bash

SYSTEM CALLS

Programming interface provided by the OS Kernel for user to access services

User

Restricted, for security

Kernel

Total privileges, access to hardwares

make System calls through OS interface:

↳ ① in user mode, program calls Kernel

② A system call is associated with a number : OS dependent

③ Each number refers to a specific service : I/O, file management, etc.

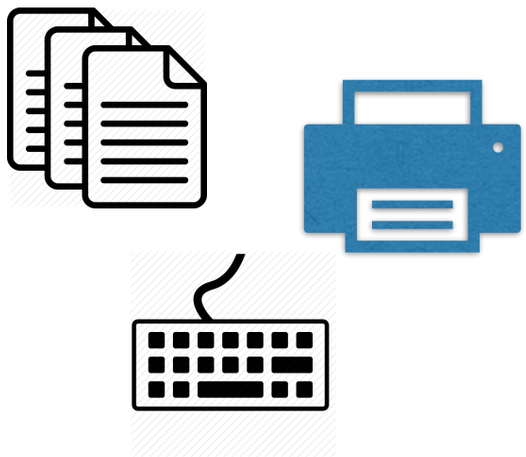
TYPES OF SYSTEM CALLS

File and device
I/O
Management

SYSTEM COMMUNICATION

PROCESS SCHEDULING

Decides which process executes, sleep, terminates, queue, priority

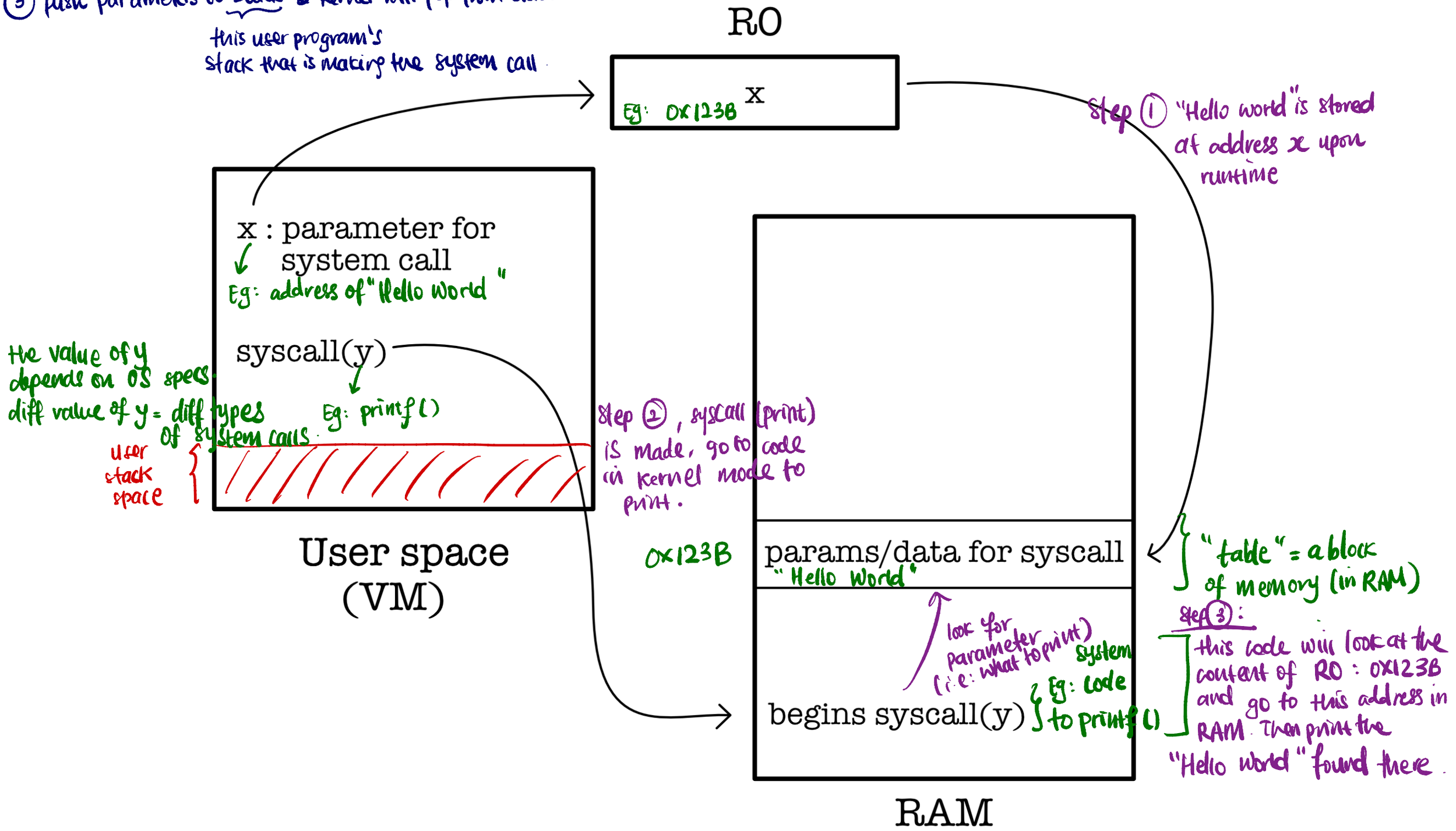


Information
maintenance
&
Security

How to pass parameters for system calls?

↳ ① parameter passing through "table"
② put parameters in the registers → limited space
③ push parameters to stack & kernel will pop from stack

this user program's stack that is making the system call



PRINT TO STDOUT

```c
#include <unistd.h>
#include <sys/syscall.h>
int main(void) {
    syscall(SYS_write, 1, "hello, world!\n", 14);
    return 0;
}
```

register # used

print

print destination, 1 means stdout

Using System Call
SYS_write is an int, OS-specific

Note: this is parameter passing via stack because "Hello world" is passed as an argument to print directly.

other eg: printf("%f is the value computed", variable)

this on the other hand is parameter passing via table because you pass the variable to print (address of it) to the print function.

```c
#include <stdio.h>
int main(void) {
    printf("hello, world!\n");
    return 0;
}
```

Using Function Call

syscall() is assembly

```
#include <sysdep.h>

/* Please consult the file sysdeps/unix/sysv/linux/x86-64/sysdep.h for
   more information about the value -4095 used below.  */

/* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5, arg6)
   We need to do some arg shifting, the syscall_number will be in
   rax.  */


        .text
ENTRY (syscall)
        movq %rdi, %rax         /* Syscall number -> rax.  */
        movq %rsi, %rdi         /* shift arg1 - arg5.  */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 is on the stack.  */
        syscall                 /* Do the system call.  */
        cmpq $-4095, %rax       /* Check %rax for error.  */
        jae SYSCALL_ERROR_LABEL /* Jump to error handler if error.  */
        ret                     /* Return to caller.  */

PSEUDO_END (syscall)
```

A BETTER CHOICE: API

Application programming interface adds layer of abstraction than direct system calls

User

Restricted, for security

API

Linux API (POSIX)
Windows API, etc

Kernel

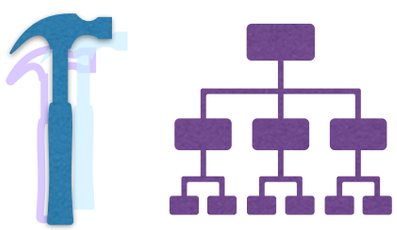
Total privileges, access to hardwares

System calls

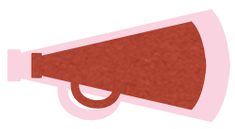
MORE ABOUT API

A good API makes it easier for developer to develop a computer program
Just like a good GUI makes it easier for user to use a computer

Software building tools



Subroutine definitions



Communication protocols

```c
void test_time(){
    // to store execution time of code
    double time_spent = 0.0;

    clock_t begin = clock();

    // do some stuff here
    sleep(3);

    clock_t end = clock();

    // calculate elapsed time by finding difference (end - begin) a
    // divide by CLOCKS_PER_SEC to convert to seconds
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("Time elapsed is %f seconds", time_spent);
}
```

And many other **function calls** to access OS services in C:

- read()
- write()
- open()
- close()
- fork()
- execvp()

Just to show you that you have been making syscalls through API more often than you thought.

SYSTEM PROGRAMS

API vs system progs : helps developer use OS services
: helps you (users) use OS services

& users (you)

A convenient environment (tools) for user applications to perform system calls and use the computer hardware

They still have to make system calls to the kernel. This means that, They are NOT run in kernel mode but they may have root access such as absolute addresses of the RAM.

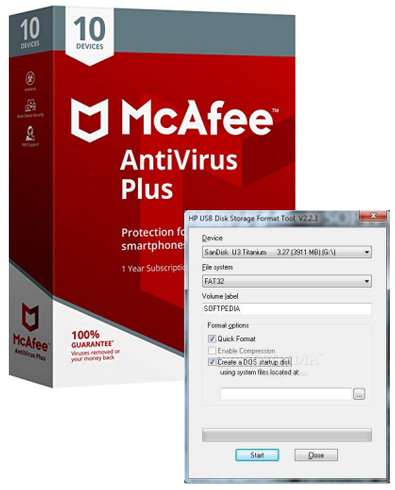
- Simple: simply UI to system calls
- Complex: perform complex series of system calls to provide services to user apps

system progs examples:

Most users' view of an OS is defined by system programs, not system call itself

- gui of desktop, terminal / cmd, antivirus, task manager / activity monitor, compiler, assembler

basically whatever comes "preinstalled" with the OS



SYSTEM PROGRAMS	USER PROGRAMS
Used for operating computer hardware	Used to perform a specific task as requested by user
Installed on the computer when OS is installed	Installed according user requirements
User doesn't typically interact with system software because they run in the background	User interacts mostly with user programs (also called application softwares)
System programs run independently means doesn't run in virtual environment because they have root access. Eg: debugger, antivirus, system accounting progs like task manager / activity monitor.	Cannot run independently, it runs in virtual environment
Provides platform for running app software	Cannot run without the presence of system programs
More example: compiler, assembler, debugger, driver, antivirus, network softwares Usually only for common prog. languages (C/C++, Java, V.Basic, PERL) We can also download compilers nowadays but these will not have root access.	Some example: media player, photos editing softwares, games

SYSTEM PROGRAM CATEGORIES

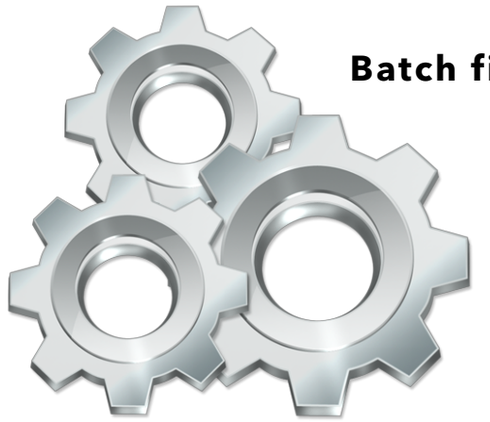
1. File manipulation
2. Status information
3. File modification
4. Programming language support
5. Program loading and execution
6. Communication
7. Application programs.
   ↳ sometimes OS comes in programs that do not need direct access to hardwares.
   For example : Notepad, Standard games, pdf reader, Statistical tools, etc

✓ Useful for program development & execution
✓ also known as "system utilities"
✓ These system program categories require "root access", meaning that they can access hardwares, especially RAM addresses directly.
✓ Note that these are not entirely run in uninterruptible kernel mode. They may still be run in user mode, but just that they are carefully written to have root access.

UI OF OS SERVICES

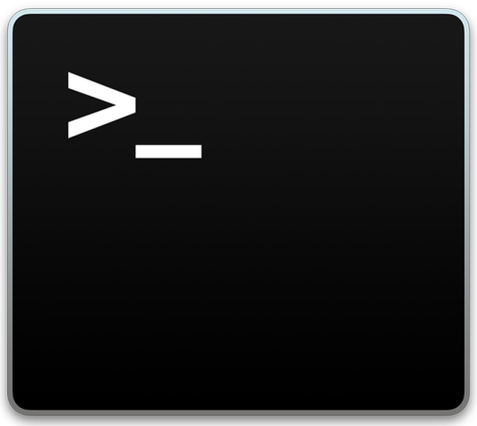
Graphical User Interface

- E.g: desktop
- Click icons to open program
- Change settings
- Close programs
- Switch between programs



Batch file

- A series of of commands: telling the computer exactly what to do
- Commands are OS-specific
- To be executed by command line interpreter
- Stored as a plaintext file

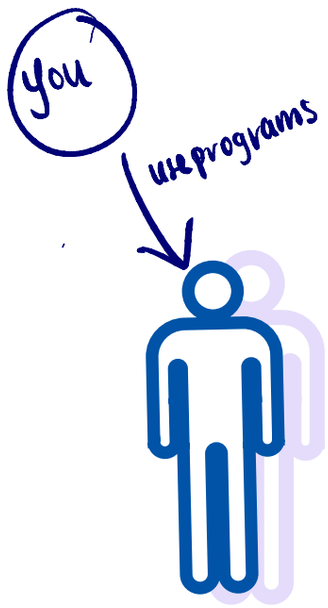


Terminal (command line interface)

- Uses shell
- A software that allows user interaction with computer via terminal
- UNIX shell: **Bash**
- Interprets commands and executes as individual processes

LAYERED APPROACH: **UNIX**

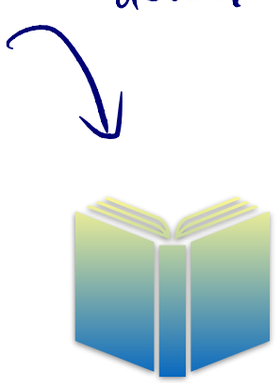
Each layer uses services provided by the layer beneath for
modularity and **ease of debugging**



User programs

High level programming language

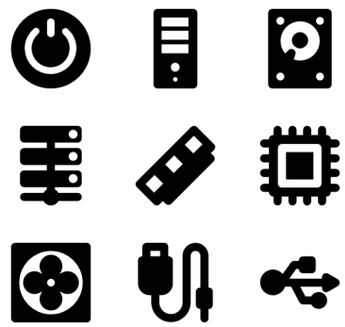
Shells, commands to run system programs: compilers, interpreters, etc



API, System Libs

System calls

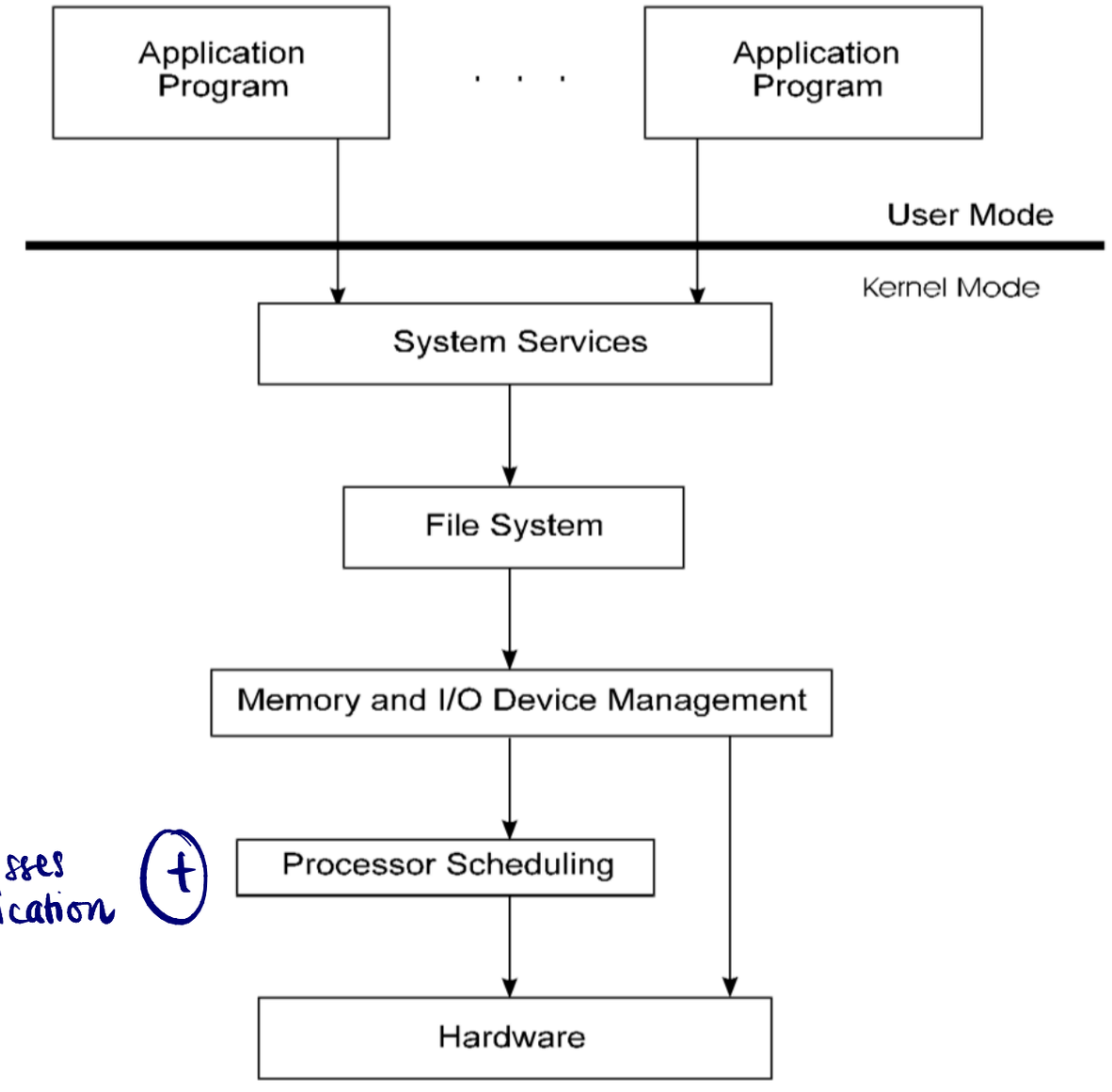
Kernel



Hardware

modularity can be achieved in 2 ways:

1. Using layers : each layer can only interact with layers above & below (O(N) connections)
2. Using modules : each module can interact with one another ($O(n^2)$ connections)

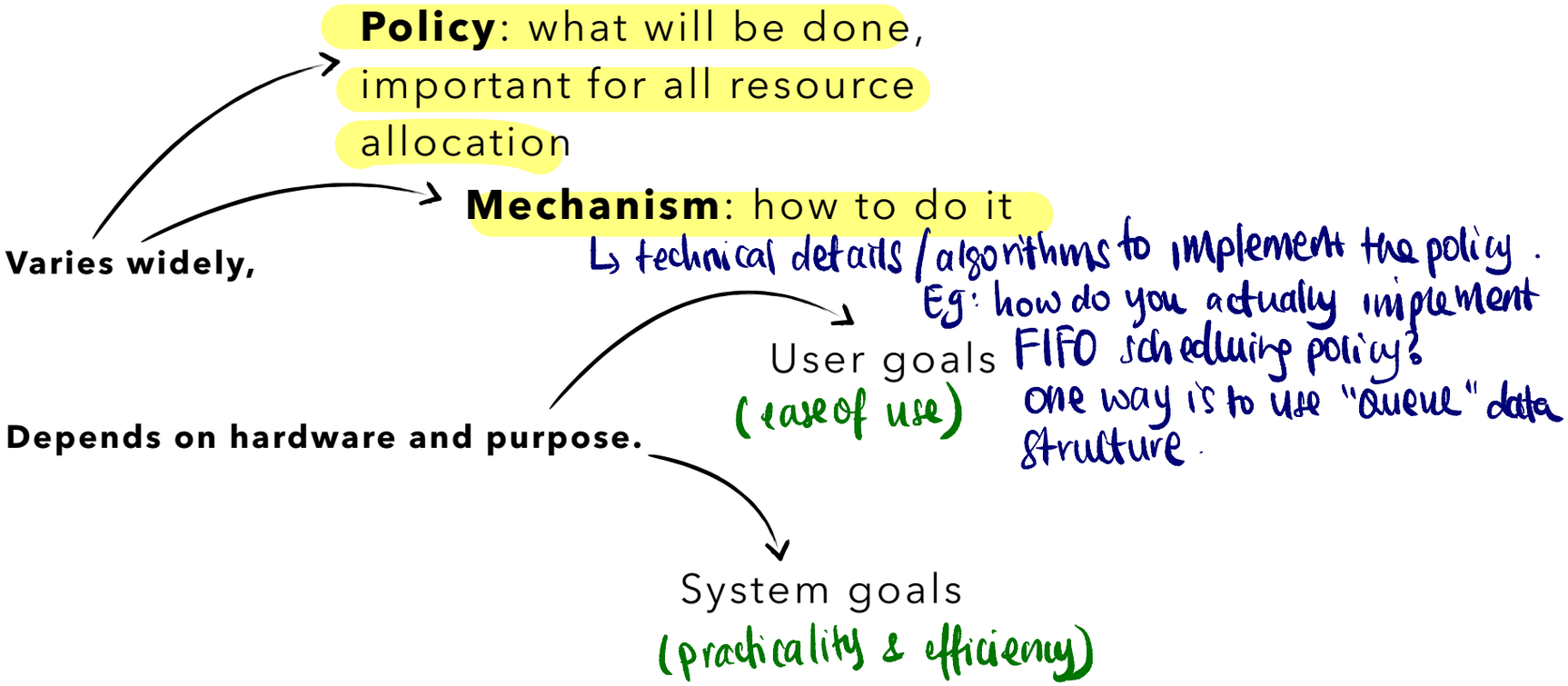


inter-processes communication (+)

Generic overview of layered approach

GENERAL IDEA OF OS DESIGN

Eg: scheduling policy, memory alloc policy, I/O interrupt handling policy.



OS EXAMPLES

Most OS is written in C + assembly.

1.Simple and layered structure (monolithic): MS-DOS, UNIX

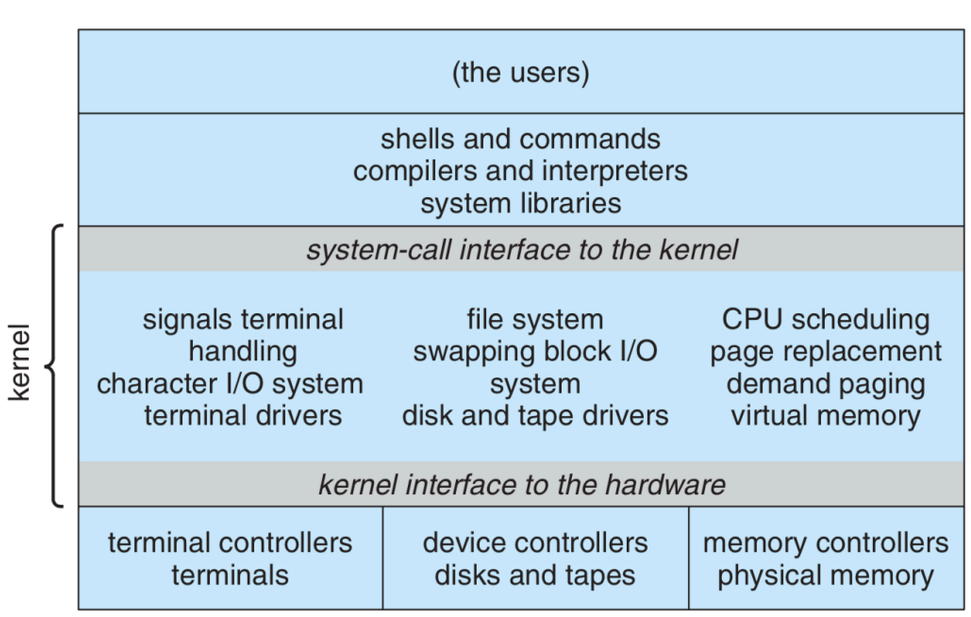


Figure 2.13 Traditional UNIX system structure.

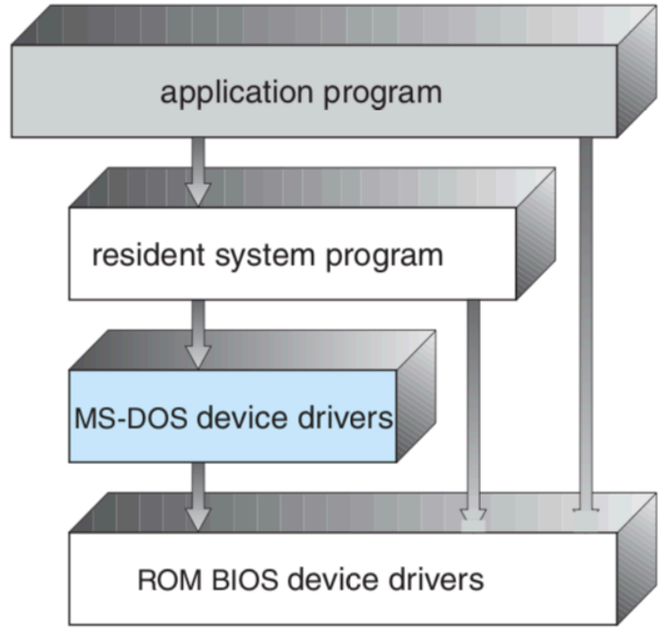


Figure 2.12 MS-DOS layer structure.

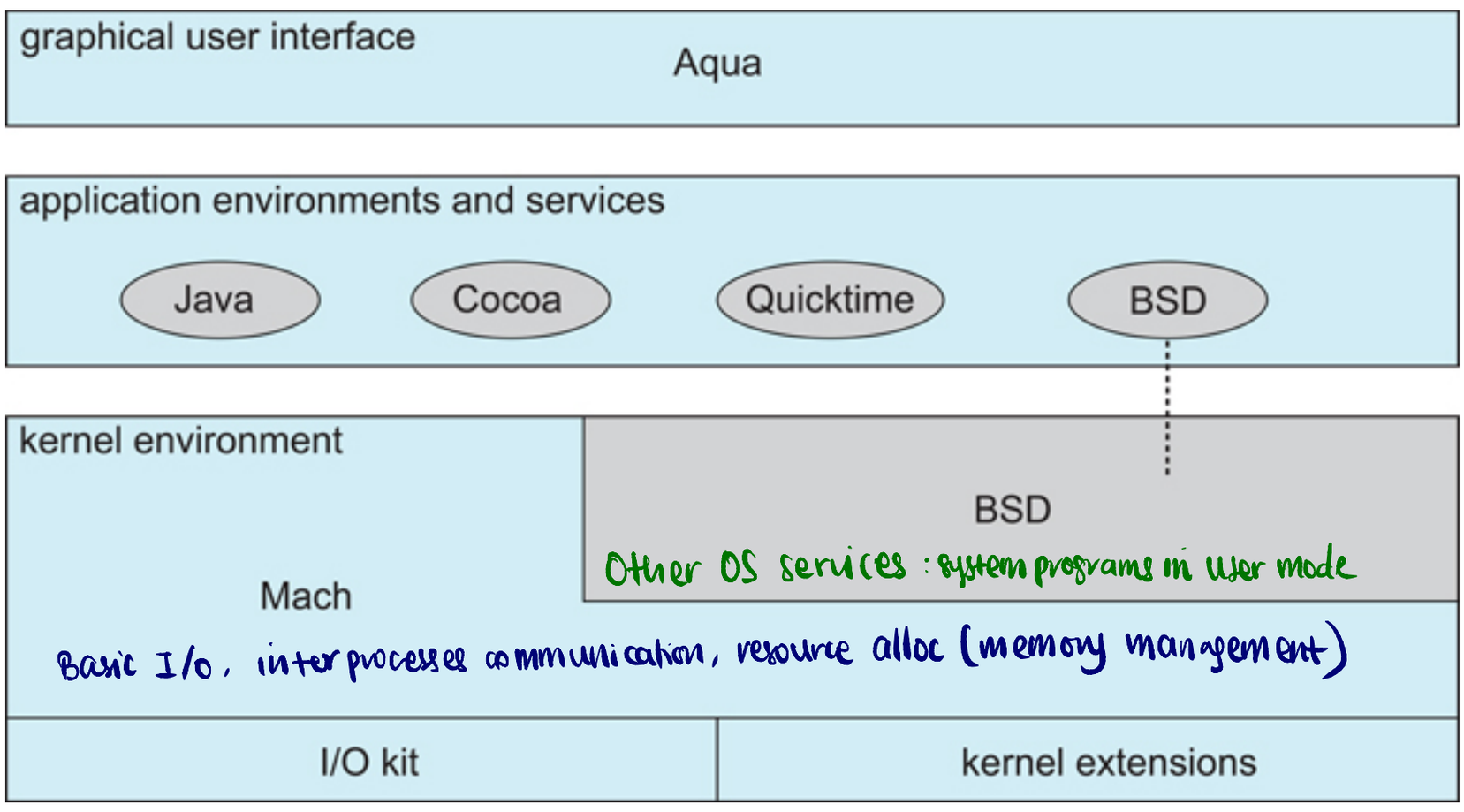
OS EXAMPLES

read: minimalist kernel

Benefits of microkernel:
1. kernel is small & lightweight (code)
2. Hence easy to debug
3. doesn't need to be updated very much since its operations are very basic & fundamental

2. Microkernels: the original Mach

Microkernel provides minimal process and memory management (1), and communication facility (2). Everything else is moved to system program and user program. (+3) some native (basic, like R/W to disk) I/O operations



OS EXAMPLES

3. Object oriented: JX - single address space system, no MMU

JX OS is mostly in Java except domainZero which is also in C + assembly. Each domain is an independent JVM.

↗ Java domains (A, B, C, etc)

An instance of JVM is made whenever a Java applet is run.

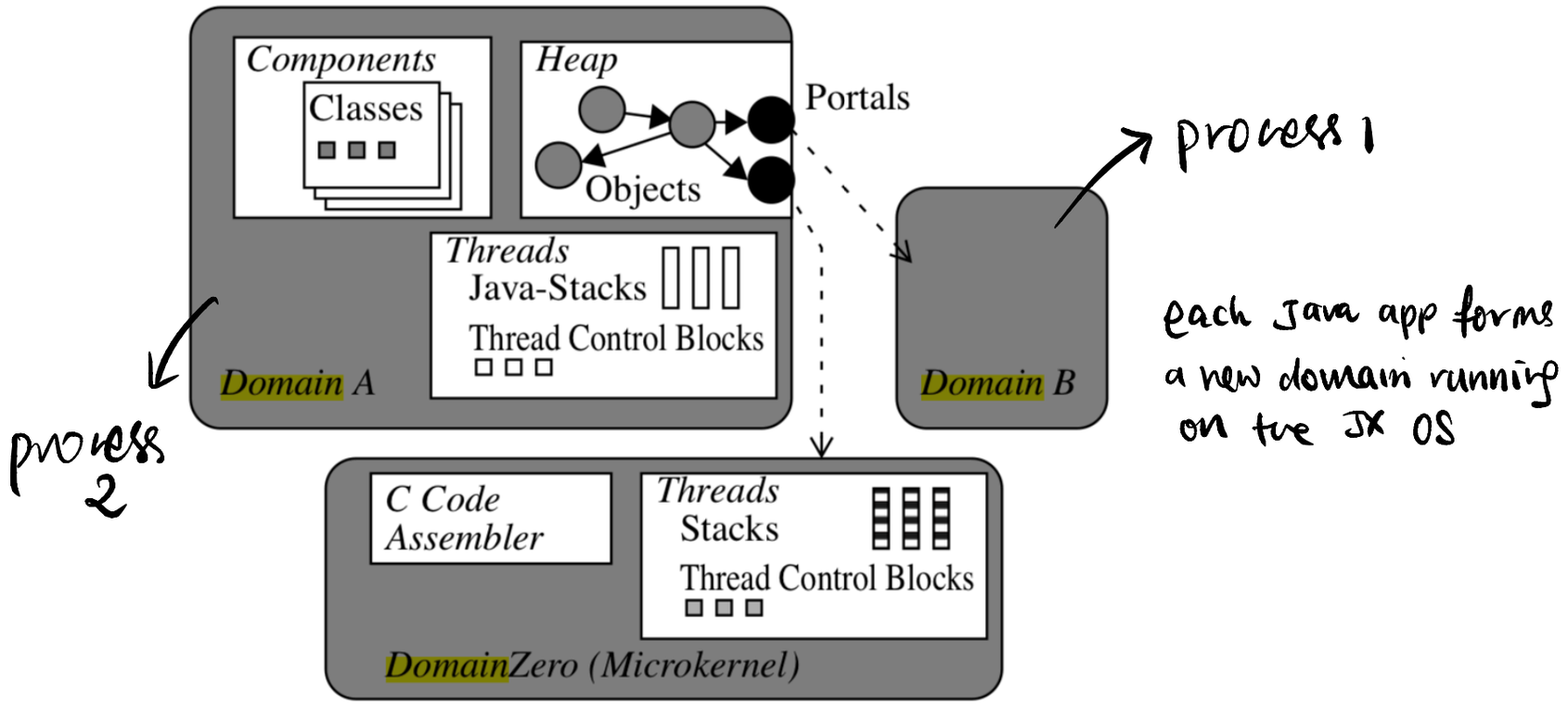
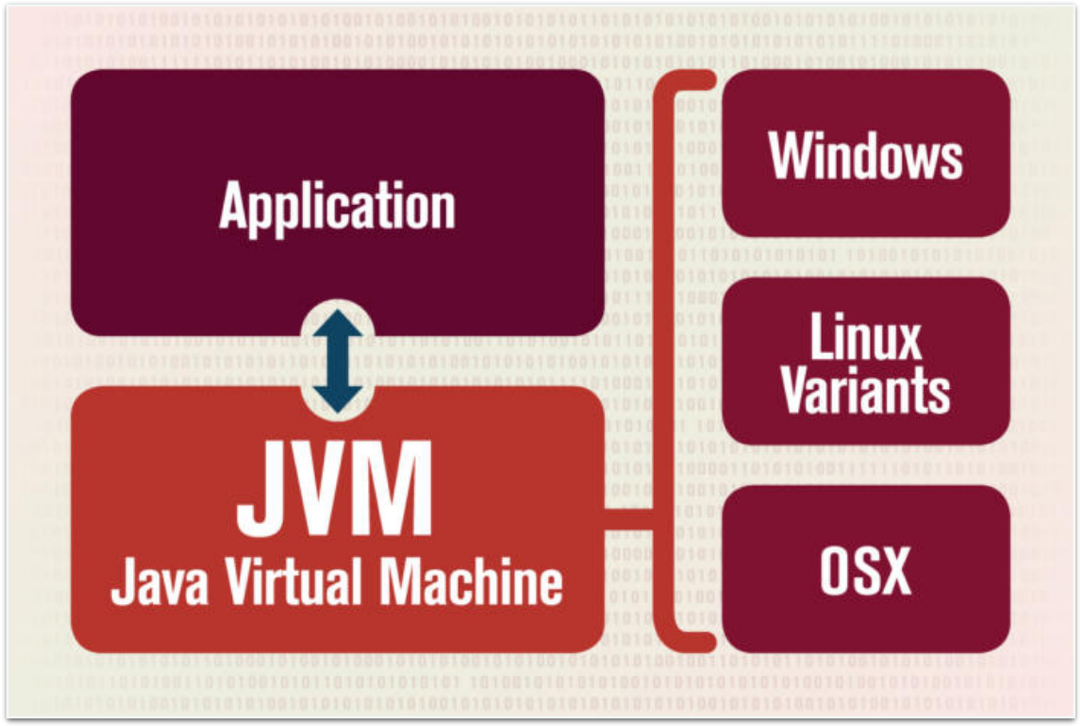


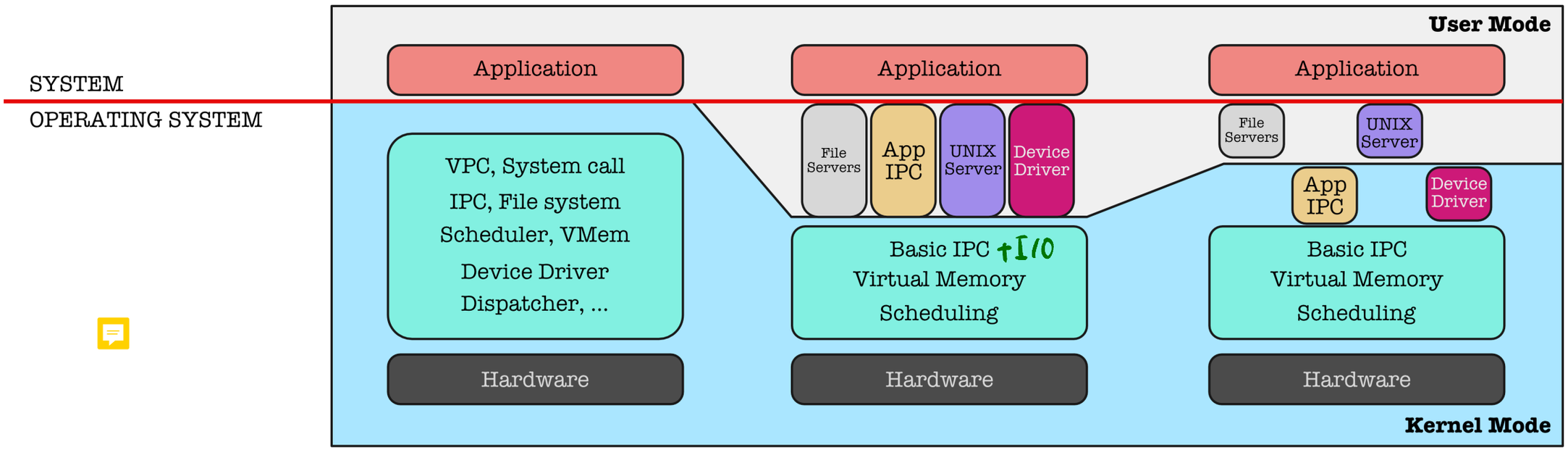
Figure 1: Structure of the JX system

MORE ABOUT JVM

JVM: An abstract machine that can run on any host OS. Takes care of its own memory management (allocation and garbage collection).



The JVM provides a **portable execution environment** for Java-based applications



Monolithic Kernel OS

layered OS, kernel does a lot of tasks & services

Microkernel OS

minimalist kernel

Hybrid OS

"medium" sized kernel.

OS STRUCTURES SUMMARY

This page onward is made for your own knowledge and is not part of the syllabus

•

LAYERED APPROACH EXTENSION: **VIRTUALIZATION**

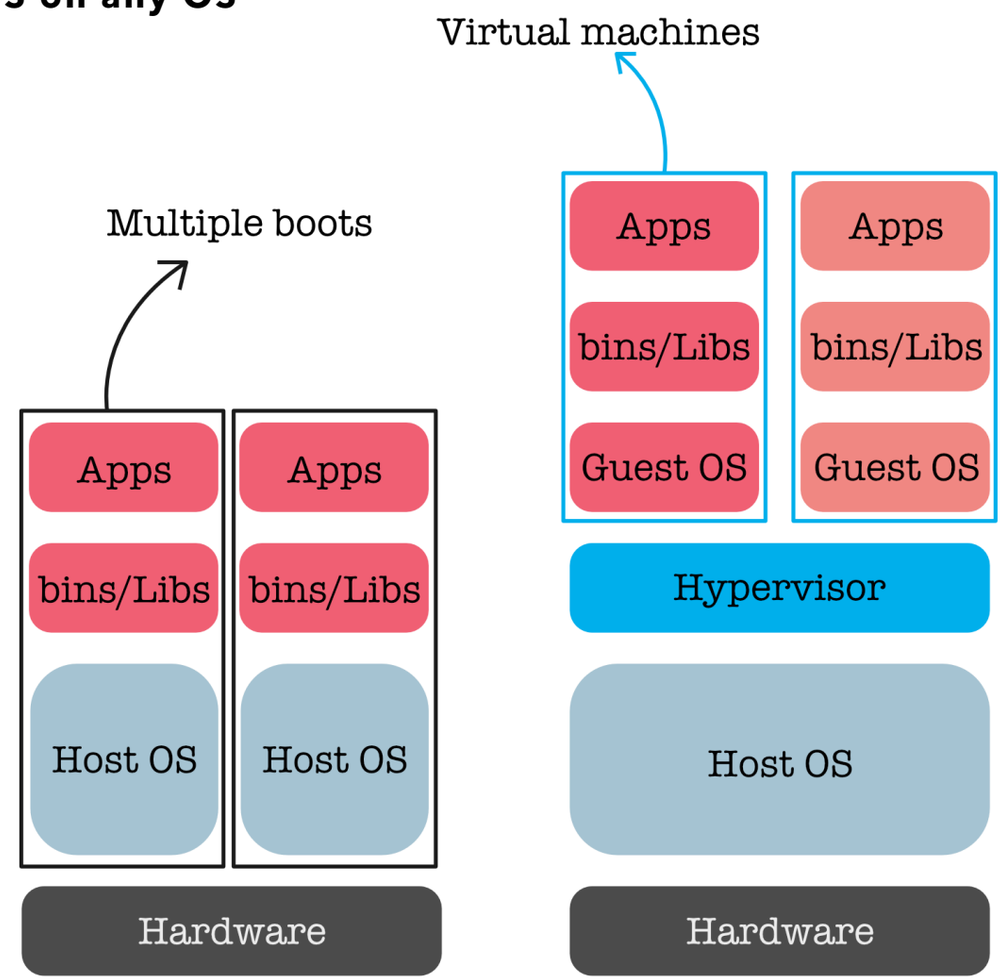
Runs any OS on any OS

- Normal machine (computers) nowadays can run any OS on its hardware, e.g: **Dual boot**

- An extension to that is **virtualization**, where you can run any OS on any OS

- Note on **hypervisor**: a computer software / firmware / hardware that runs virtual machines.

- Examples of hypervisors: VMWare, VMWorkstation, VirtualBox, Parallel Desktop for Mac, etc



Normal Machine **Virtualization**

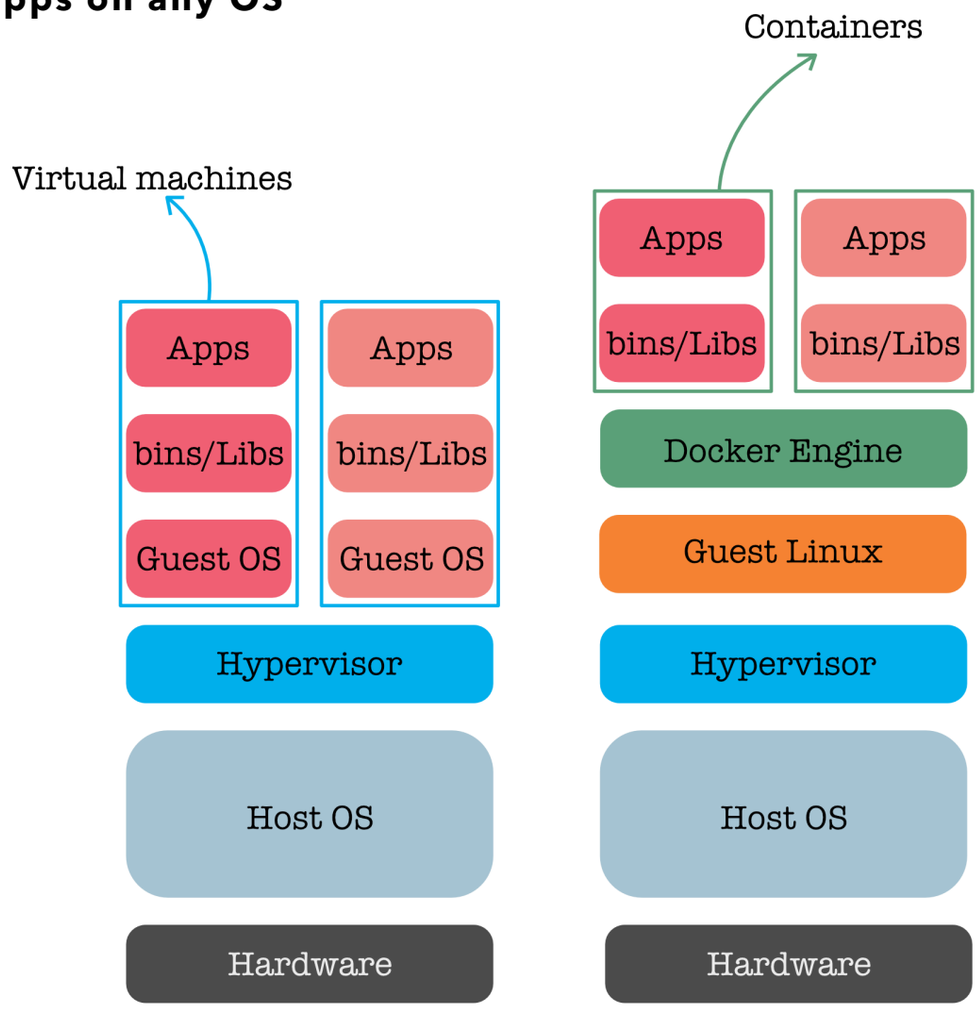
LAYERED APPROACH EXTENSION:
CONTAINERIZATION

Runs any apps on any OS

- A further extension to this is **containerisation**

- **Containers** allow a developer to package up an application with **all** of the parts it **needs**, such as libraries and other dependencies, and ship it all out as one package

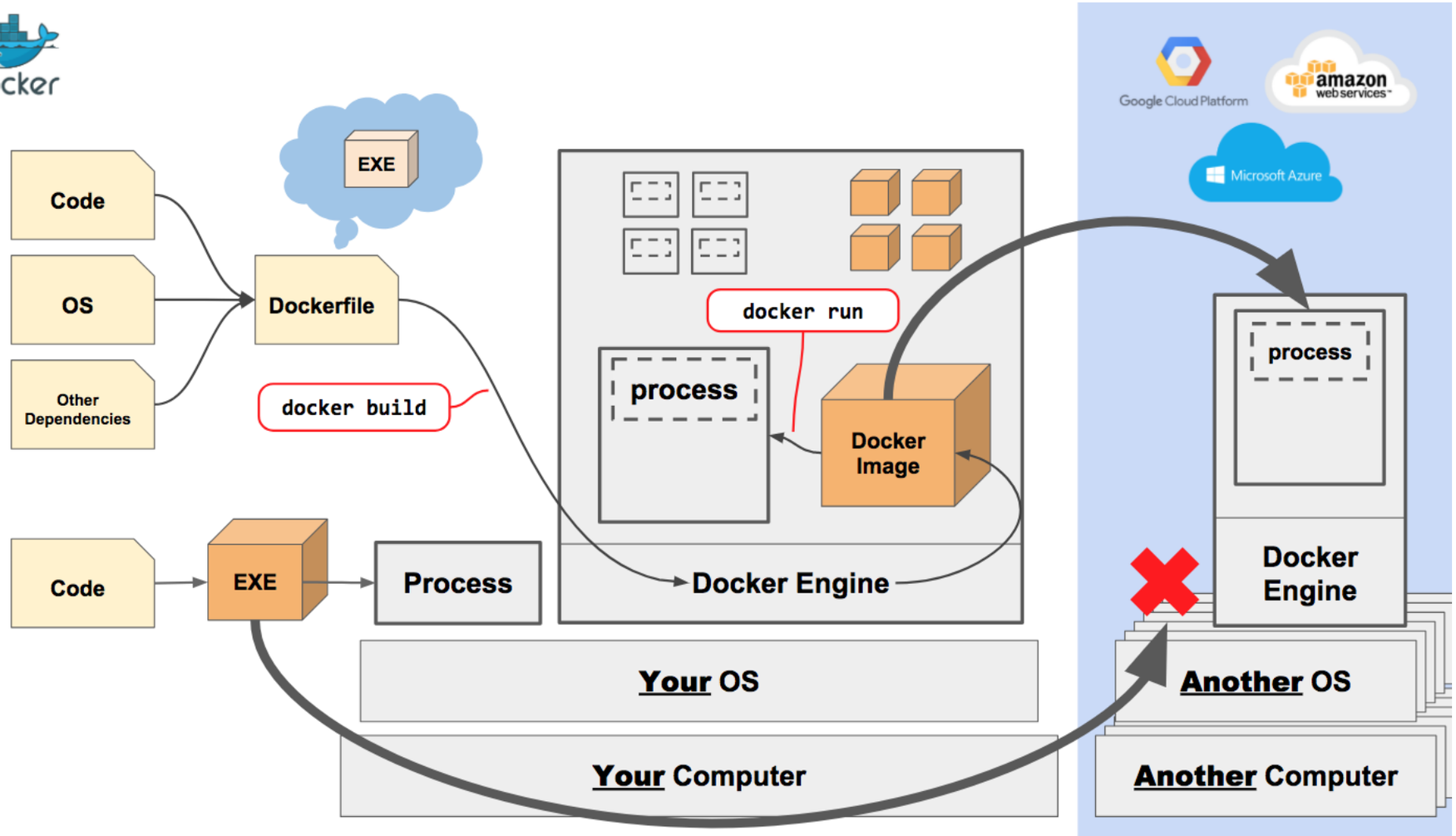
- Example of softwares that support containerisation: **Docker**



Virtualization **Containers**

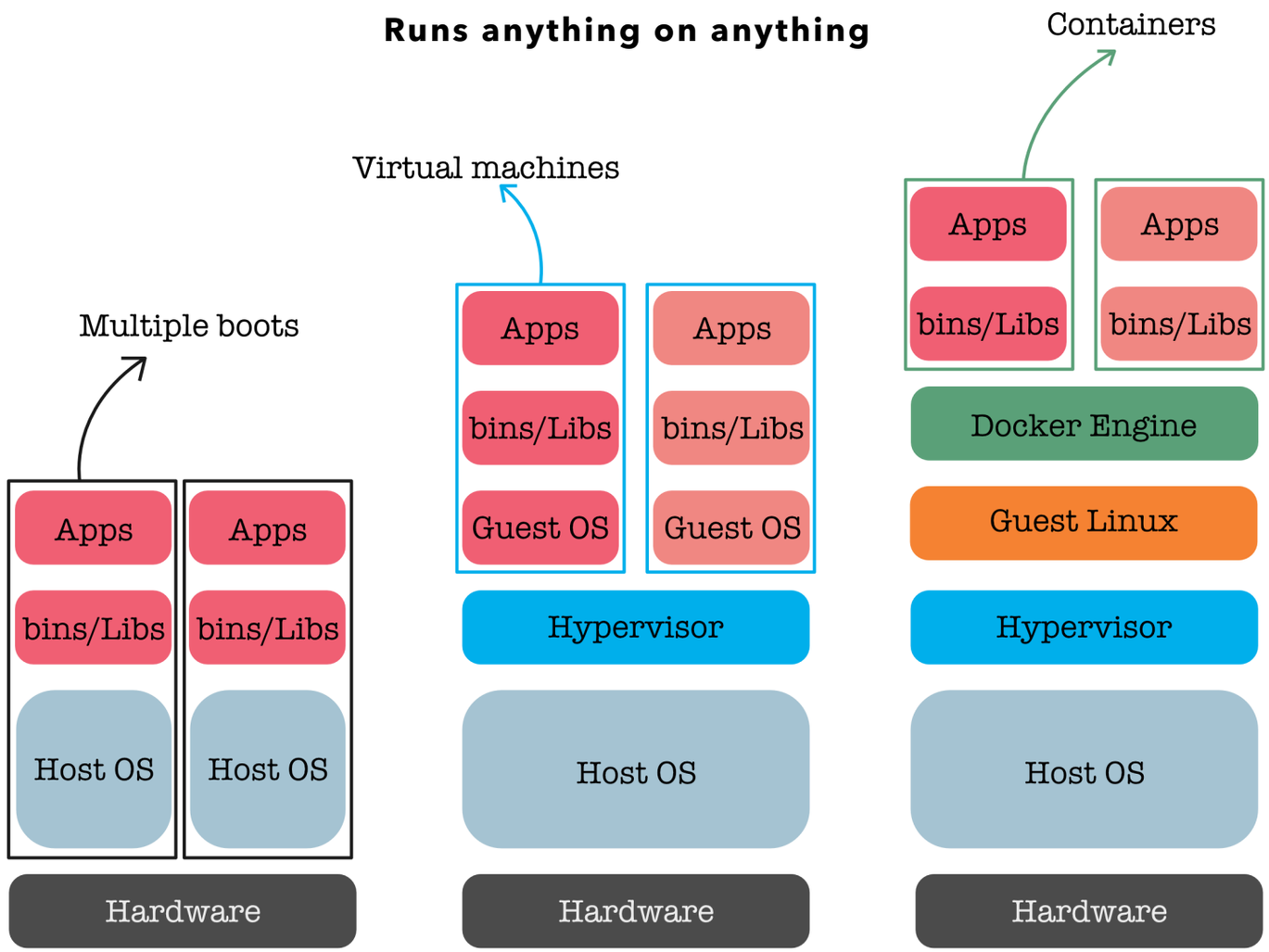
DOCKER

https://medium.freecodecamp.org/docker-quick-start-video-tutorials-1dfc575522a0



LAYERED APPROACH ALLOWS

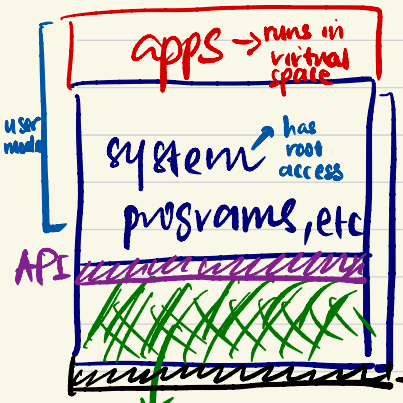
ABSTRACTION



Summary

Structure of OS (architecture)

1. monolithic { contains layers
   many services offered in kernel mode
2. microkernel { IPC & I/O (basic), resource allocation, program exec in kernel mode
   small kernel, easy to maintain
3. object oriented



user apps relies on services provided by OS. They make system call to request these services

3 ways to pass parameters:
1. by table
2. direct to Registers
3. by stack

OS services:
1. I/O
2. info/security maintenance
3. process exec & scheduling
4. system & device management

kernel:

- the part of the OS that has full authority of the hardware.
- Runnig at all times
- Uninterruptible
- contains implementation of system calls