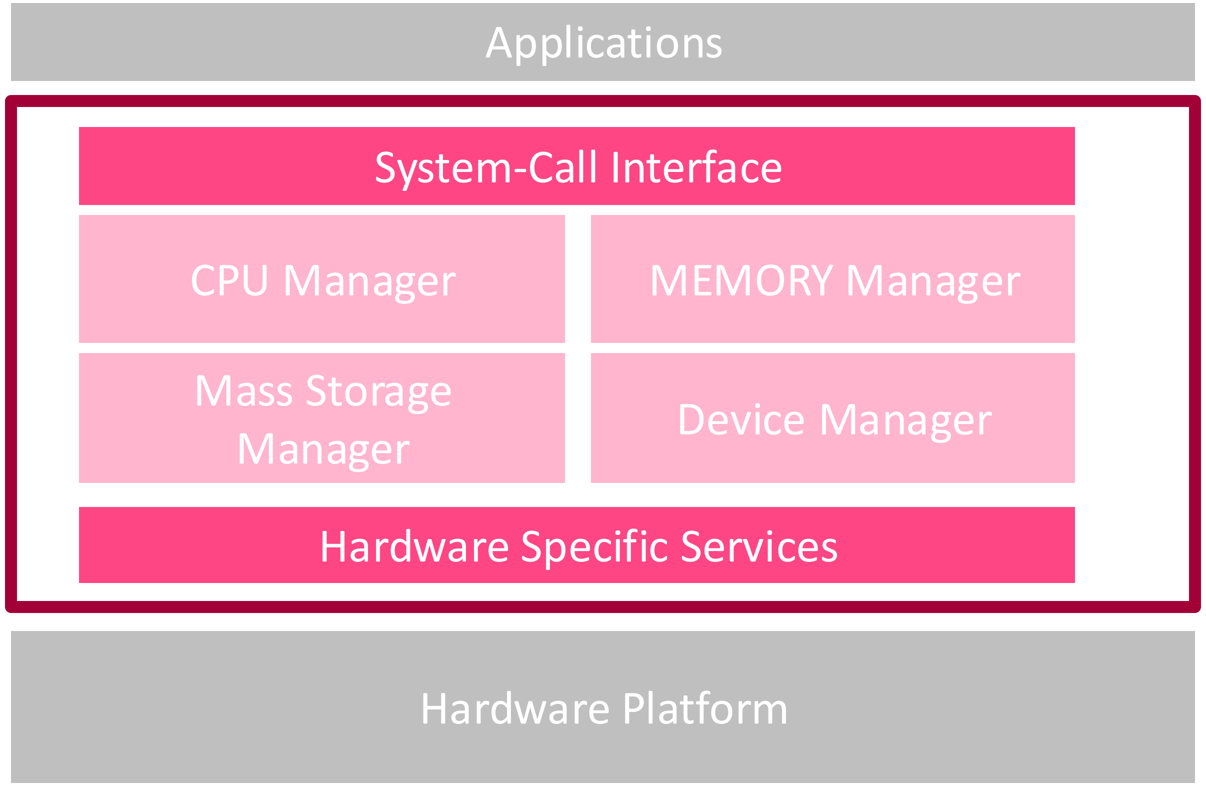
INTRODUCTION TO OPERATING SYSTEMS

STEFANO DI CARLO

OPERATING SYSTEM DEFINITION

- An Operating System (OS) is a **System Software** that **manages computer** hardware and software **resources** and **provides services** to **users** **programs**.
- It acts as an intermediary between users and the hardware of a computer.



A VIEW OF OPERATING SYSTEM SERVICES

User and other system programs

GRAY DOMAIN

GUI | Command Line | Touch Screen

User Interfaces

OS DOMAIN

System calls

Program Execution | I/O operations | File systems | Communic. | Resource allocation | Logging

Error detection/ Corrxection | Protection and Security

OPERATING SYSTEM SERVICES

- Operating systems provide an environment for execution of **programs** and services to **programs** and **users**
- Services helpful to the user:
  - **User interface** — Command-Line (CLI), Graphics User Interface (GUI), touch-screen, Batch
  - **Program execution** — The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)

OPERATING SYSTEM SERVICES

- Services helpful to the programs:
  - **I/O operations** — A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** — The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Communications** — Processes may exchange information, on the same computer or between computers over a network

OPERATING SYSTEM SERVICES

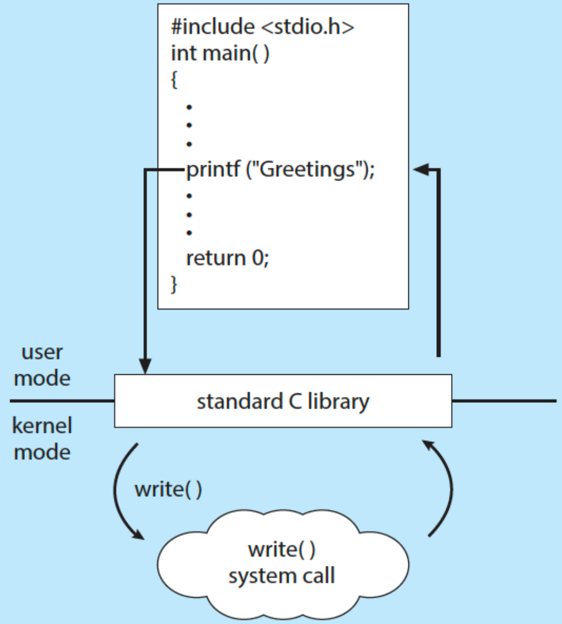
- Functions for ensuring the efficient operation of the system via resource sharing
  - **Resource allocation** — When multiple users or multiple jobs run concurrently, resources must be allocated to each of them.
  - **Logging** — To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** — The owners of information stored in a multiuser or networked computer systems may want to control use of that information, concurrent processes should not interfere with each other
    - Protection involves ensuring that all access to system resources is controlled
    - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
  - **Error detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

SYSTEM CALLS

- A **system call** is a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- A system call is a way for programs to **interact with the operating system**.
- A computer program makes a system call when it requests the operating system's kernel.

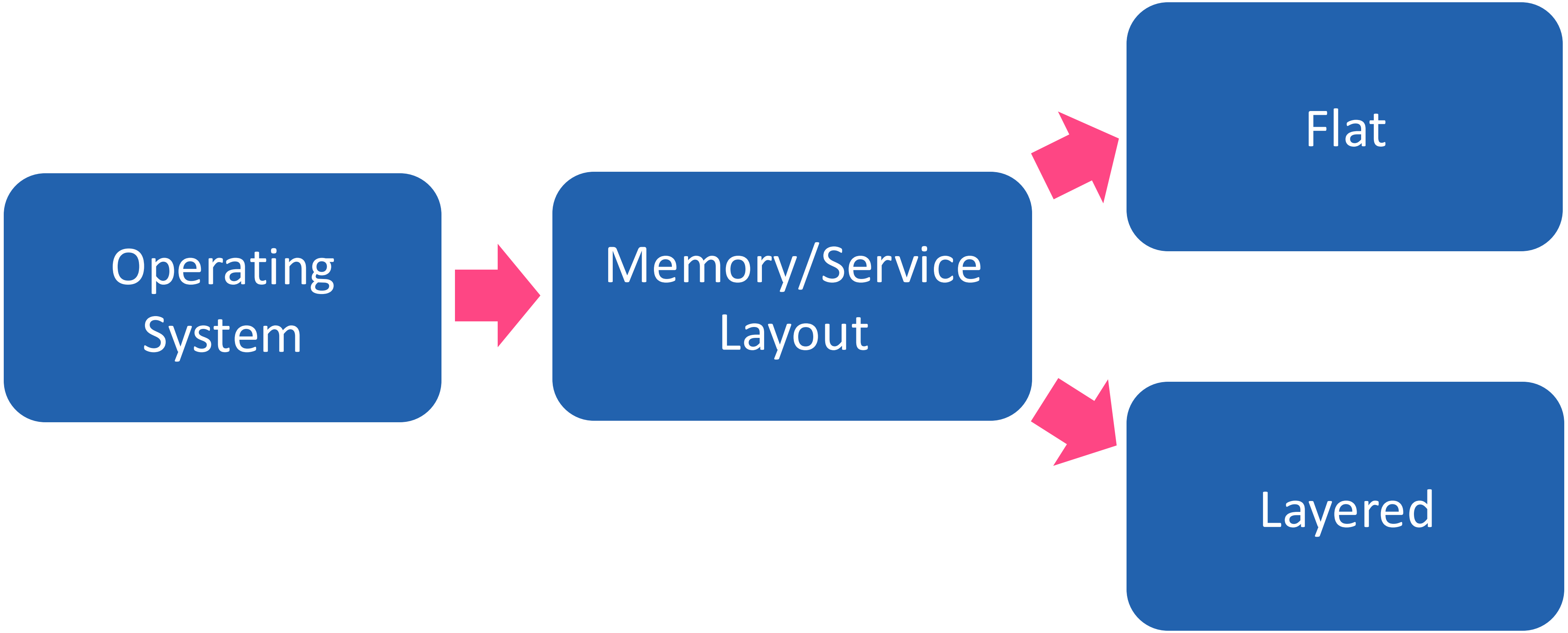
THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



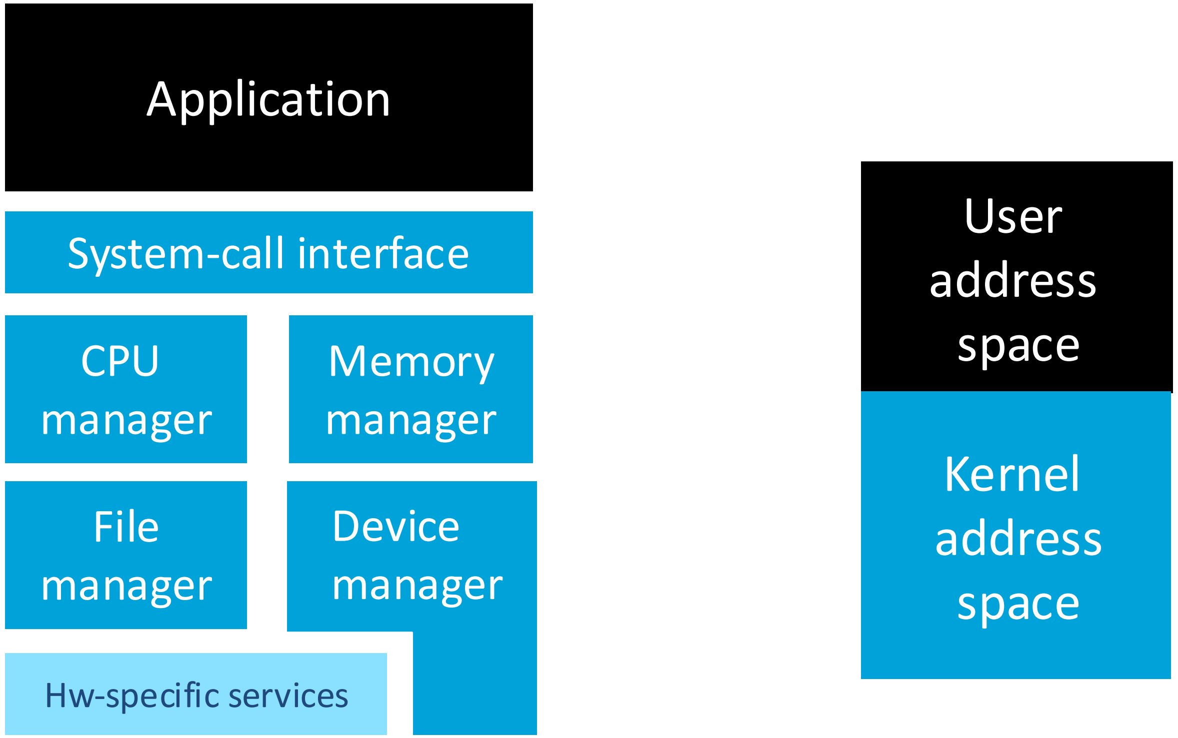
[Taken from Operating Systems 10th Edition — Silbershatz, Galvin and Gagne © 2018]

OPERATING SYSTEMS ARCHITECURES



FLAT ARCHITECTURE

- No strict memory separation between application and operating system
- Intended to provide most of the functionalities in the smallest space with minimum hardware support
- The components of the operating system are essentially functions that any application can invoke
- Examples
  - FreeRTOS
  - Micrium mC/OS
  - MS-DOS
  - FreeDos



FLAT ARCHITECTURE

OS BUILD PROCESS

Syscall.c

```
void sys_write (…) {
...
}

Int sys_read (…) {

}
```

scheduler.c

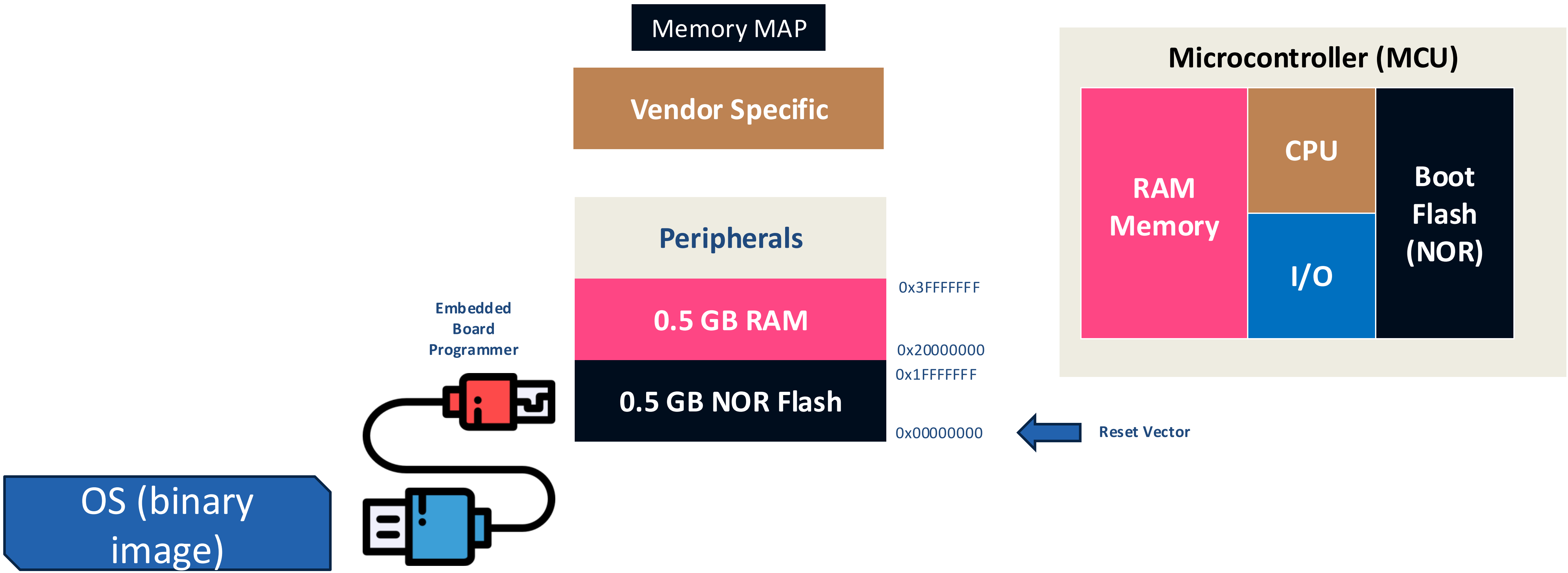
```
void scheduler (…) {
...
}
```

main.c

```
int main (…) {
    hw_init();
    os_init();
    …
    scheduler();
    …
    while(1);
}
```

compile → link → OS (binary image, e.g., elf)

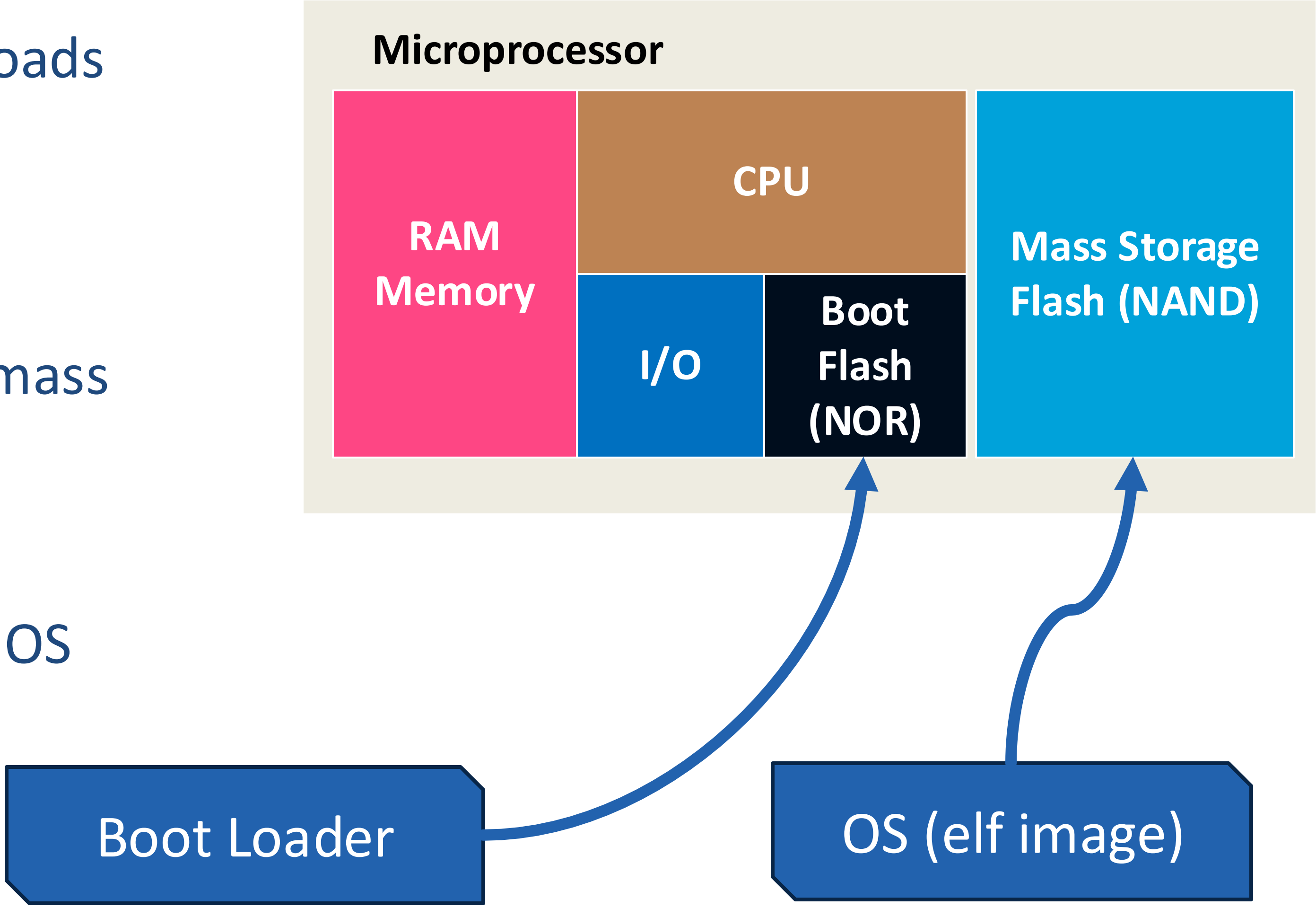
HOW AN OS RUN?


Memory MAP
Vendor Specific
Peripherals
0.5 GB RAM
0.5 GB NOR Flash
0x3FFFFFFF
0x20000000
0x1FFFFFFF
0x00000000
Reset Vector
Embedded Board Programmer
OS (binary image)
Microcontroller (MCU)
RAM Memory
CPU
I/O
Boot Flash (NOR)

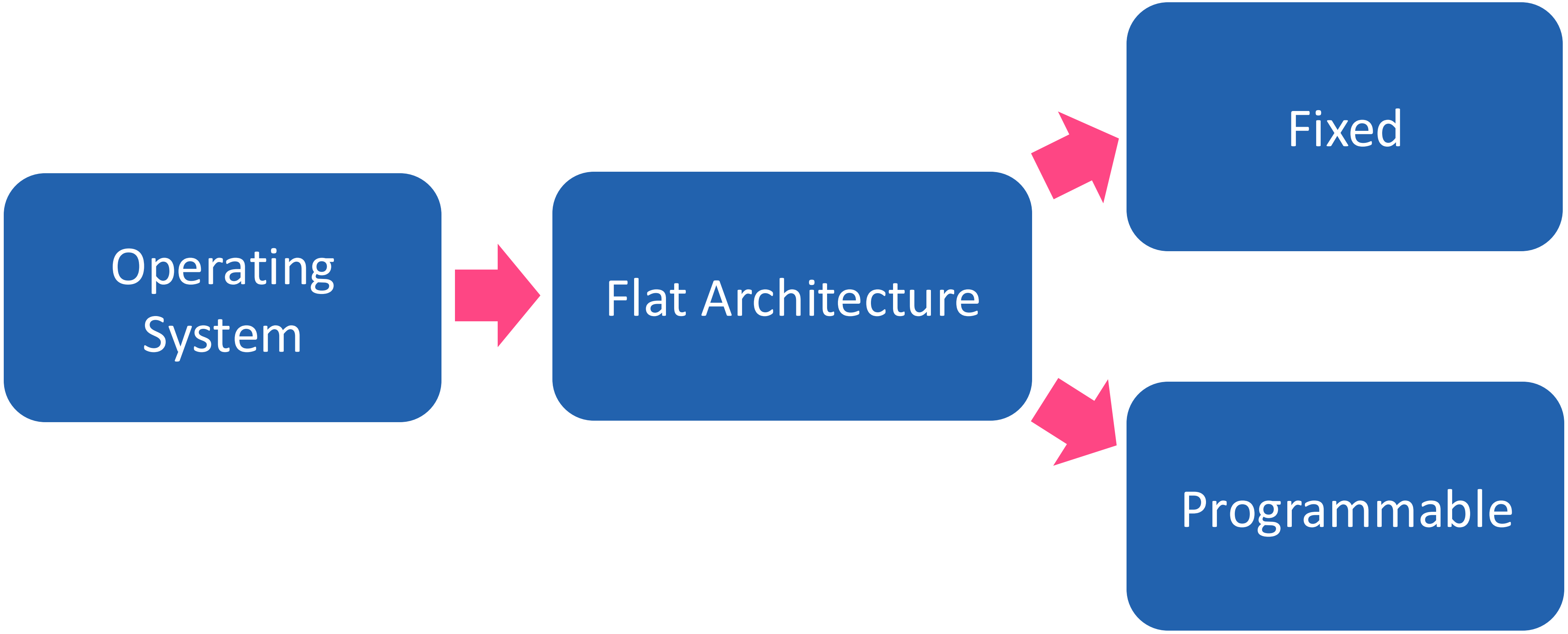
HOW AN OS RUN?

A bootstrap **loader** is a **program** that loads the operating system or runtime environment for the computer after completion of self-tests

1. Initialize essential hardware (e.g., mass storage flash)
2. Load the OS image in RAM
3. Jump to the first instruction of the OS



WHAT ABOUT APPLICATION PROGRAMS?



FIXED TASKS

- No need to change the build and run model

Syscall.c

```
void sys_write (…) {
...
}

Int sys_read (…) {

}
```

scheduler.c

```
void scheduler (…) {
...
}
```

scheduler.c

```
int main (…) {
      hw_init();
     os_init();
     scheduler();
     …
     while(1);
}
```

tasks.c

```
int task1 (…) {
     …
}
Int task2 () {
     …
}
```

compile → link → OS (binary image)

PROGRAMMABLE TASKS

OS BUILD PROCESS

Syscall.c

```
void sys_write (…) {
. . .
}

Int sys_read (…) {

}
```

scheduler.c

```
void scheduler (…) {
. . .
}
```

scheduler.c

```
int main (…) {
    os_init();
    …
    scheduler();
    …
    while(1);
}
```

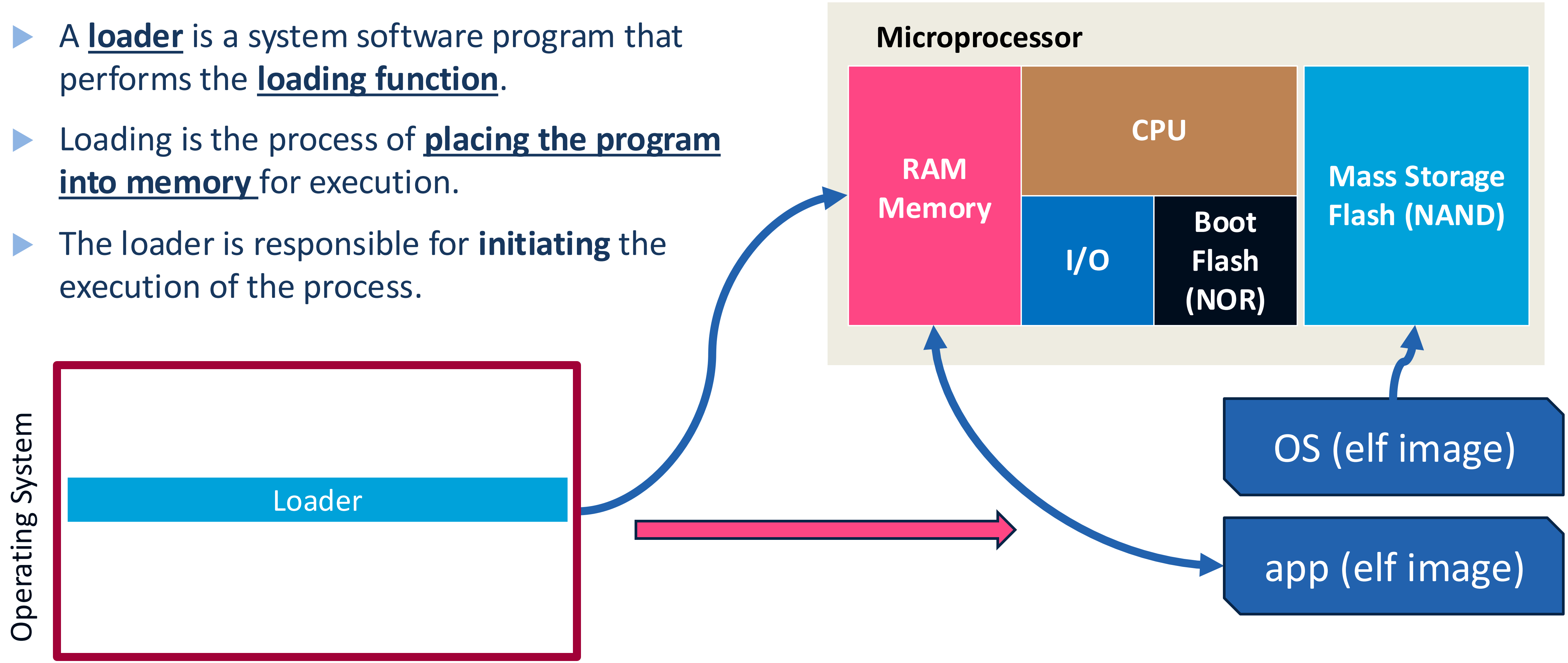
loader.c

```
int loader (…) {
    …
}
```

compile → link → OS (elf image)

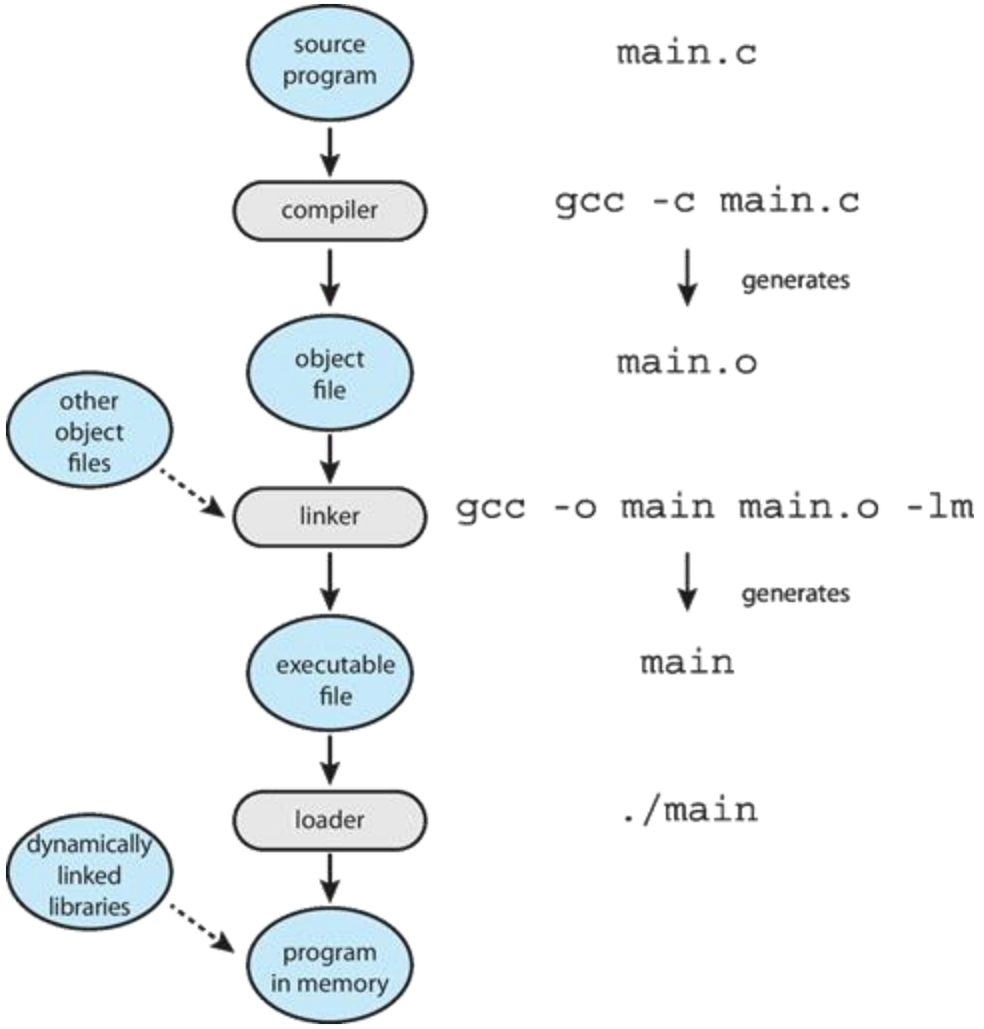
LOADER

- A **loader** is a system software program that performs the **loading function**.
- Loading is the process of **placing the program into memory** for execution.
- The loader is responsible for **initiating** the execution of the process.



LINKERS AND LOADERS

- Source code compiled into object files designed to be loaded into any physical memory location – relocatable object file
- Linker combines these into single binary executable file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by loader to be executed
  - Relocation assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general-purpose systems don't link libraries into executables
  - Rather, dynamically linked libraries (in Windows, DLLs) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them

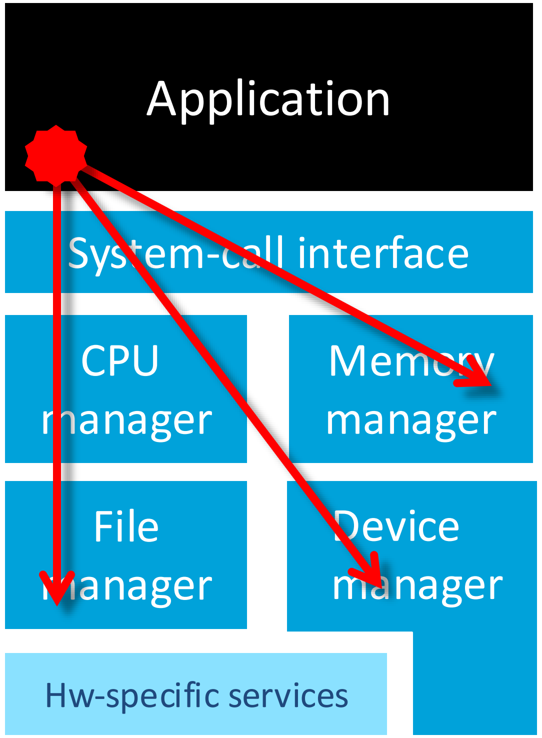


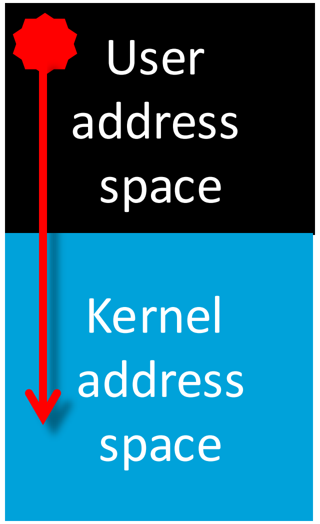
[Taken from Operating Systems 10th Edition — Silbershatz, Galvin and Gagne © 2018]

FLAT ARCHITECTURE

- No strict memory separation between application and operating system
- Intended to provide most of the functionalities in the smallest space with minimum hardware support
- The components of the operating system are essentially functions that any application can invoke
- Examples
  - FreeRTOS
  - Micrium mC/OS
  - MS-DOS
  - FreeDos

- Malfunctions can freely propagate corrupting the system

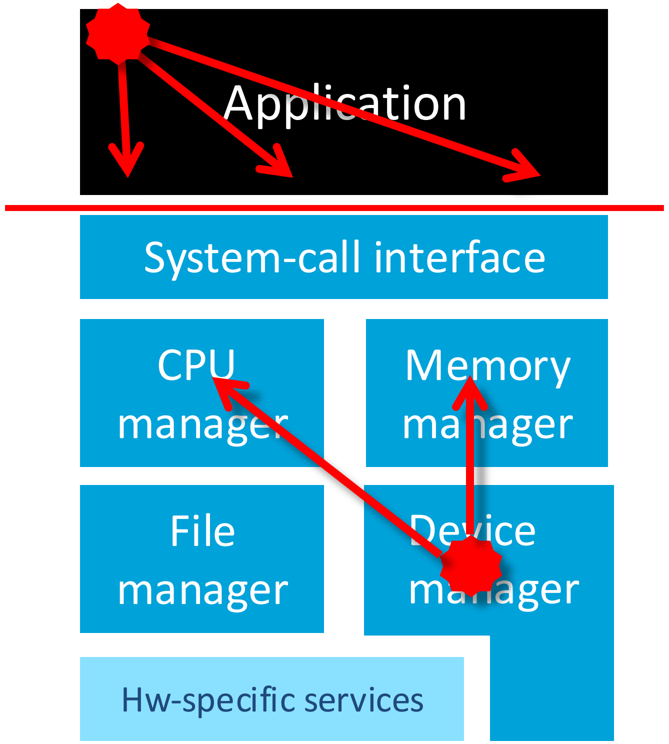


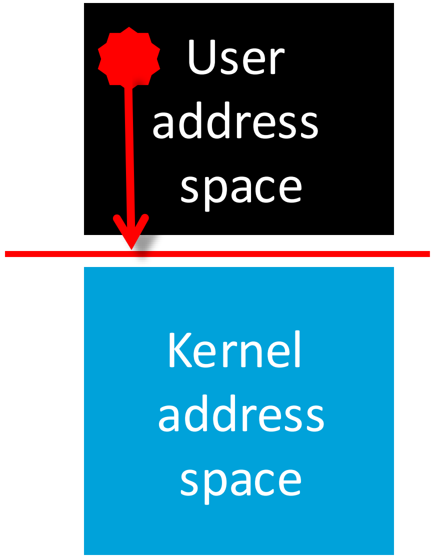


MONOLITHIC KERNEL

- The computing architecture is split into two separated domains
  - <u>User space</u>**:** running application and systems programs
  - **<u>Kernel space:</u>** The OS kernel including everything below the system-call interface and above the physical hardware
- There is separation between kernel memory and user memory
  - They require additional hardware support such as MMU, MPU and CPU operating modes
- Examples
  - Linux

- **<u>Malfunctions in the application cannot propagate to the kernel</u>**



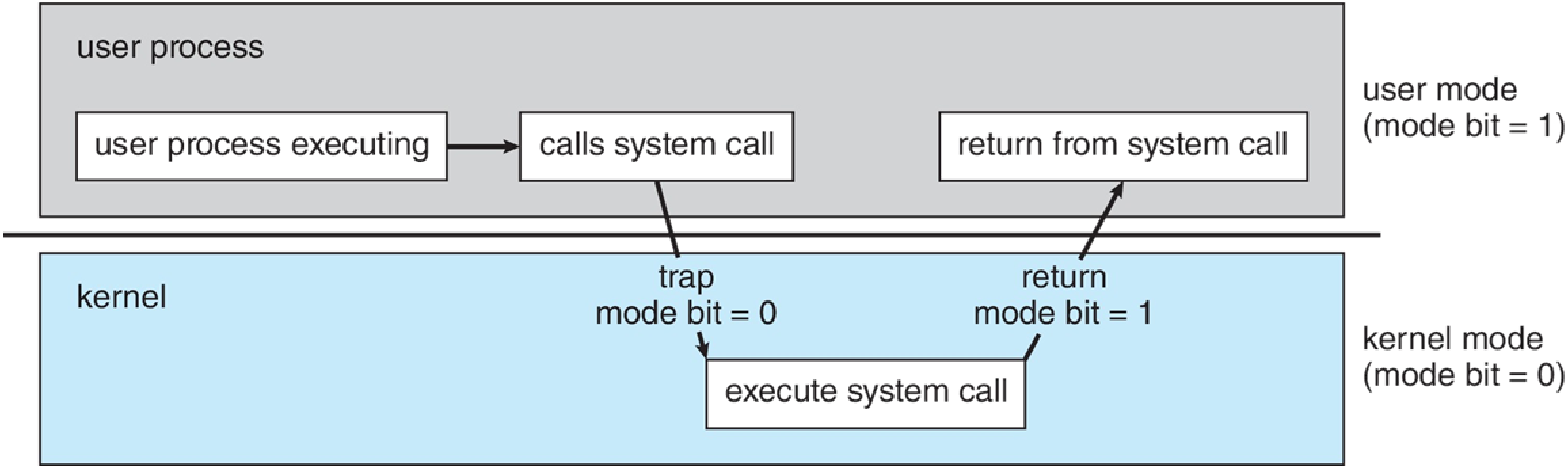


MONOLITHIC KERNEL

- No protection between operating systems components
  - Faulty drivers can crash the whole system
  - More than 2/3 of today OS code are drivers
- Few figures
  - Drivers cause 85% of Windows XP crashes
  - Error rate in Linux drivers is 3x than in other part of the Kernel
- Causes for driver bugs:
  - 23% programming errors
  - 38% mismatch regarding device specification
  - 39% OS/Driver interface misconception

MONOLITHIC KERNEL

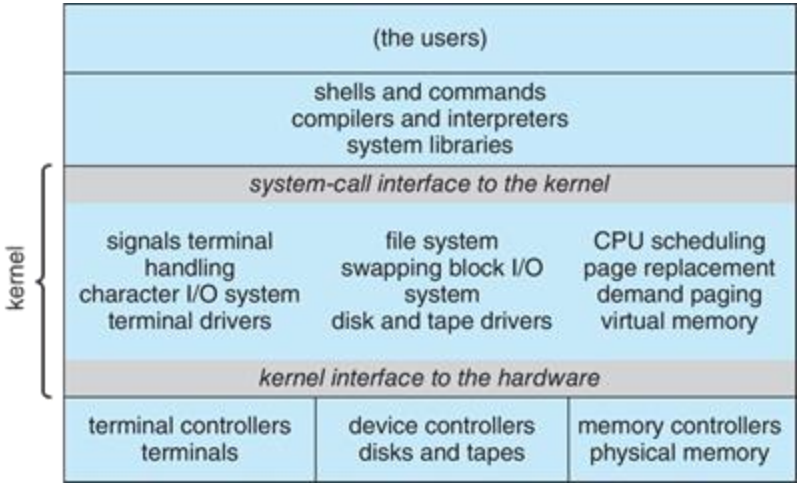
- User space and kernel space execution benefit from the availability of different execution modes in the CPU



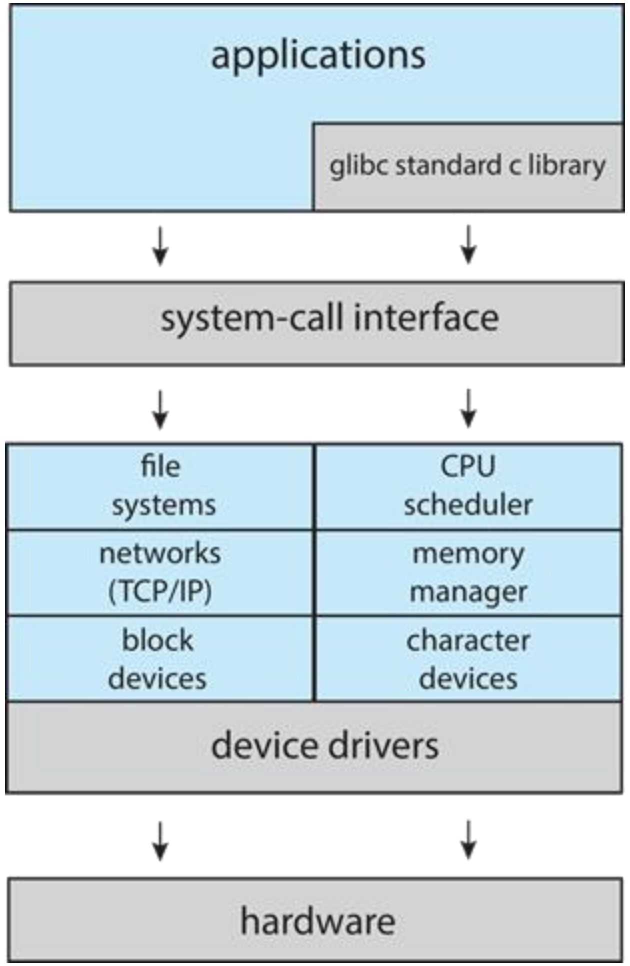
[Taken from Operating Systems 10th Edition — Silbershatz, Galvin and Gagne © 2018]

MONOLITHIC KERNEL EXAMPLES

Traditional Unix Architecture



Linux Architecture (monolithic + modules)



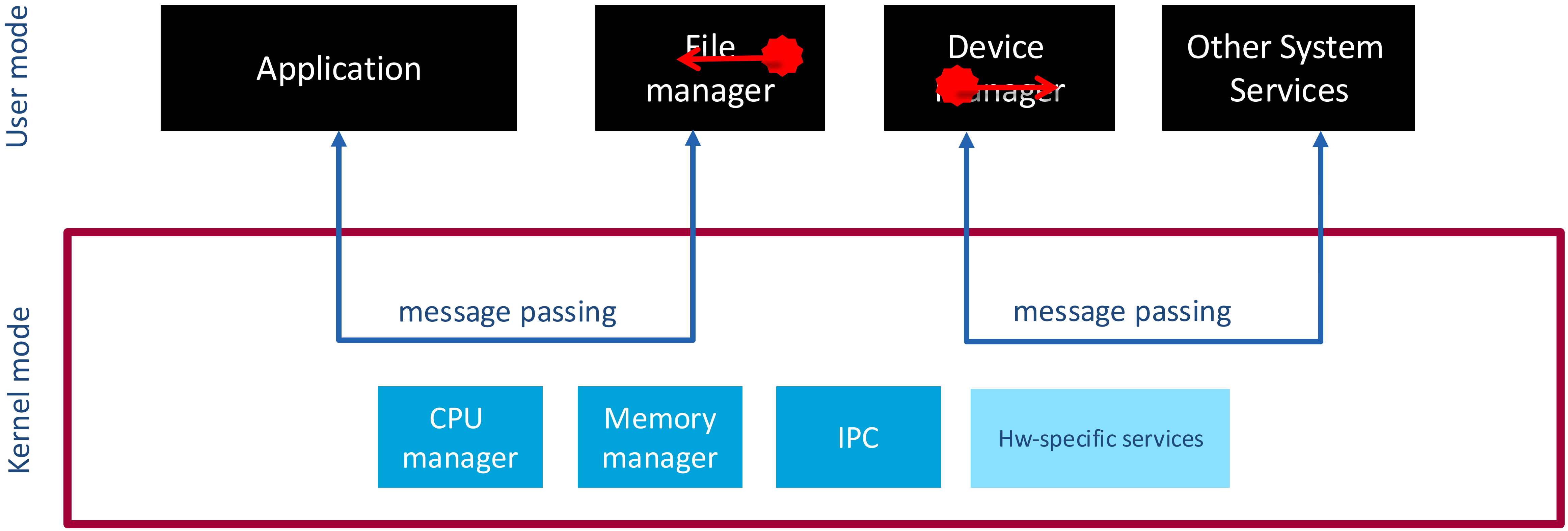
[Figures Taken from Operating Systems 10th Edition — Silbershatz, Galvin and Gagne © 2018]

MICROKERNELS

- **Moves as much from the kernel into user space**
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

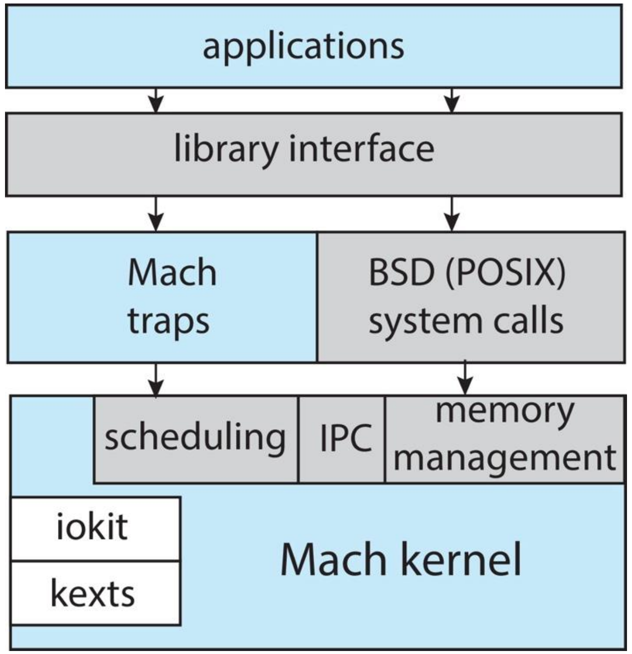
MICROKERNEL SYSTEM STRUCTURE

- Malfunctions in the user space cannot corrupt the whole system

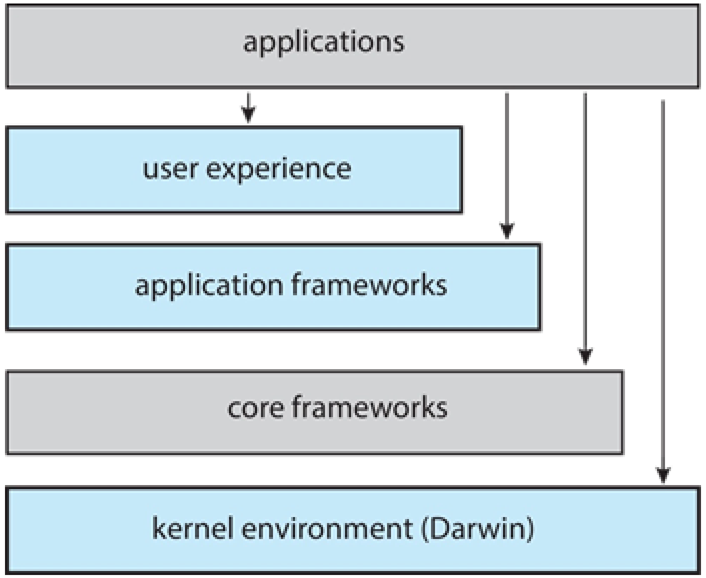


MICROKERNEL EXAMPLES

Darwin (Mac OSX)



IOS



[Figures Taken from Operating Systems 10th Edition — Silbershatz, Galvin and Gagne © 2018]

MONOLITHIC KERNELS VS MICROKERNELS

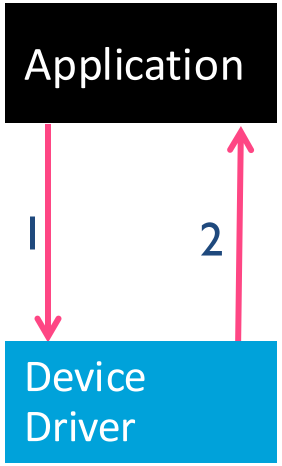
- Microkernels can be better validated than monolithic kernel as much smaller
  - Less code to read and checks, easier to guarantee the correctness of the code

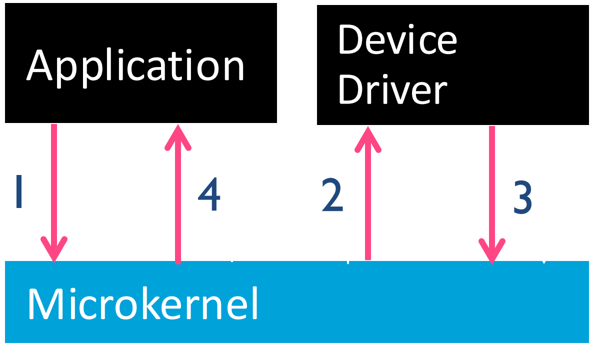
- Example: i386
  - L4 microkernel: 15.000 lines of code
  - Linux: 300.000 lines of code excluding drivers

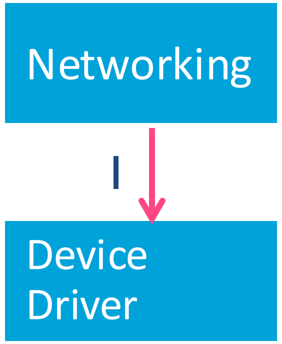
- Monolithic kernels have better performance in
  - Executing system calls
  - Calls between operating system components

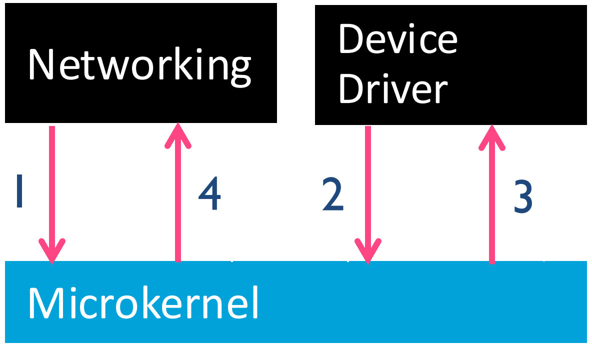
MONOLITHIC KERNELS VS MICROKERNELS

- System call performance
  - Monolithic kernel: 2 context switches
  - Microkernel: 4 context switches
- Calls between operating system components
  - Monolithic kernel: 1 function call
  - Microkernel: 4 context switches









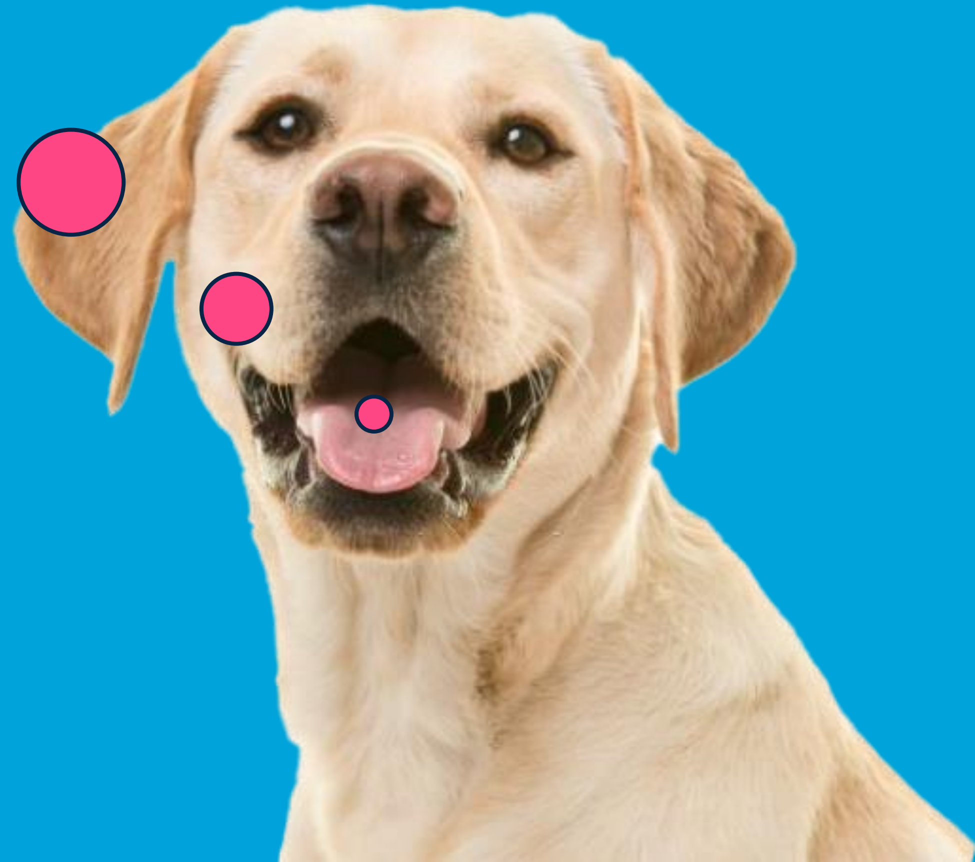
HYBRID SYSTEMS

- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem personalities
- Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)

LET'S TRY TO WRITE A TOY FLAT OS

- https://baltig.polito.it/teaching-material/exercises-caos-and-os/myfirstos

QUESTIONS?



THANK YOU!