

# Introduction to Operating Systems

M1 MOSIG – Operating System Design

Renaud Lachaize

# Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - **Many slides directly adapted from those of the CS140 class**
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Textbooks (Silberschatz et al., Tanenbaum)

# Course goals

- **Introduce you to operating system concepts**
  - Hard to use a computer without interacting with the OS
  - Understanding the OS makes you a better (more effective) programmer
- **Cover important system concepts in general**
  - Caching, concurrency, memory management, I/O, protection, ...
- Teach you to deal with larger software systems
- Prepare you to take other classes related to OS concepts
  - M1 Principles of computer networks, M1/M2 Distributed systems, M1 Parallel systems, M2 Cloud infrastructure, M2 Virtualization & advanced OS, ...

# Outline

- What is an operating system?
- Some history
- Abstractions: processes and address spaces
- Protection and resource management

# What is an operating system?

- *An operating system (OS) is a (software) layer between the hardware and the applications*
- Two key roles: **virtualization** and **resource management**

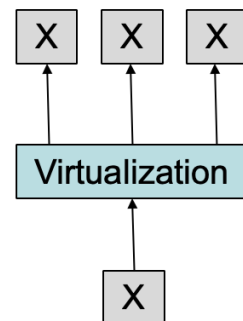
# What is an operating system?

- **Virtualization**

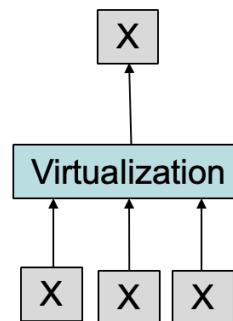
- *The OS makes it easier to write and run programs on a machine*

- Hides the low-level interface of the hardware and replaces it with higher-level abstractions
- Hides the physical limitations of a machine and the differences between machines (size of the main memory, number of processors)
- Hides the sharing of resources between applications/users

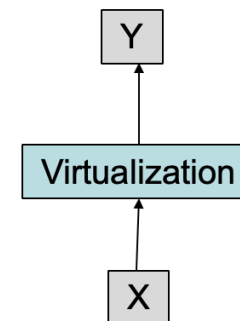
- Thus, we sometimes refer to the OS as a “*virtual machine*”



Multiplexing



Aggregation



Emulation

# What is an operating system? (continued)

- **Resource management**

- The OS is in charge of managing the resources of a computer system
  - **Physical resources:** memory, processor, devices, ...
  - **Logical resources:** programs, data, communications, ...
- Goals: allow the applications to run safely / securely / efficiently / fairly ... despite the fact that they run concurrently
- Encompasses several dimensions, including: ***allocation, sharing*** and ***protection***
- Consists in a combination of ***mechanisms*** and ***policies***

# OS Design goals and trade-offs

- Provide **useful abstractions** to improve programmer/administrator/user productivity
- Provide high **performance**
  - Leverage the power/capacity of the hardware
  - Minimize the (time and space) overhead of the OS features
- Provide **protection**
  - Between applications
  - Between applications and OS
  - Between users
- Provide a high degree of **reliability**
- Take care of other aspects such as predictability, energy-efficiency, ...



# OS Interfaces

- An operating system typically exports **two kinds of interfaces**
  - A command/user interface
  - A programmatic interface
- **Command/user interface**
  - Designed for human users
  - Various forms: textual or graphical
  - Composed of a set of commands
    - Textual example (Unix shell): `rm myfile.txt`
    - Graphical example (most systems): drag the myfile.txt icon into the trash bin.

# OS Interfaces (continued)

- **Programmatic interface**

- This interface is used/called from application programs running on the system
  - Including the programs implementing command/user interfaces
- Composed of a set of procedures/functions
  - **Libraries**
  - **System calls** (more details later)
- Defined both:
  - At the source code level: ***Application Programming Interface (API)***
  - At the machine code level: ***Application Binary Interface (ABI)***

Some of the topics that we will study during the semester

- **How does the OS virtualize and manage resources?**
  - What are the required mechanisms and policies?
  - What kind of support is required from the hardware?
  - How can these goals be achieved efficiently?
  - **We will consider several resources** : CPU, main memory, input/output (I/O) devices (e.g., storage devices)
- **How to build concurrent programs?**
  - How to program applications with several “tasks”?
  - How to coordinate these tasks and let them share data?
  - How to make such programs correct and efficient?
  - What kind of support is needed from the OS and the hardware to achieve this goals?

# Outline

- What is an operating system?
- **Some history**
- Abstractions : processes and address spaces
- Protection and resource management

# Some History

## (1) Early operating systems

- In the beginning, **the OS did not do much**
- Essentially, a set of libraries of commonly-used functions (e.g., low-level code for I/O devices)
- **Running one program at a time**
- Possibly involving a human operator (e.g., for deciding in what order to run the jobs)
- **Assumed no bad users or programs**
- **Problem: poor utilization**
  - ... of hardware (e.g., CPU idle while waiting for I/O completion)
  - ... of human user (must wait for each program to finish)

# Some History

## (2) Beyond libraries: Protection

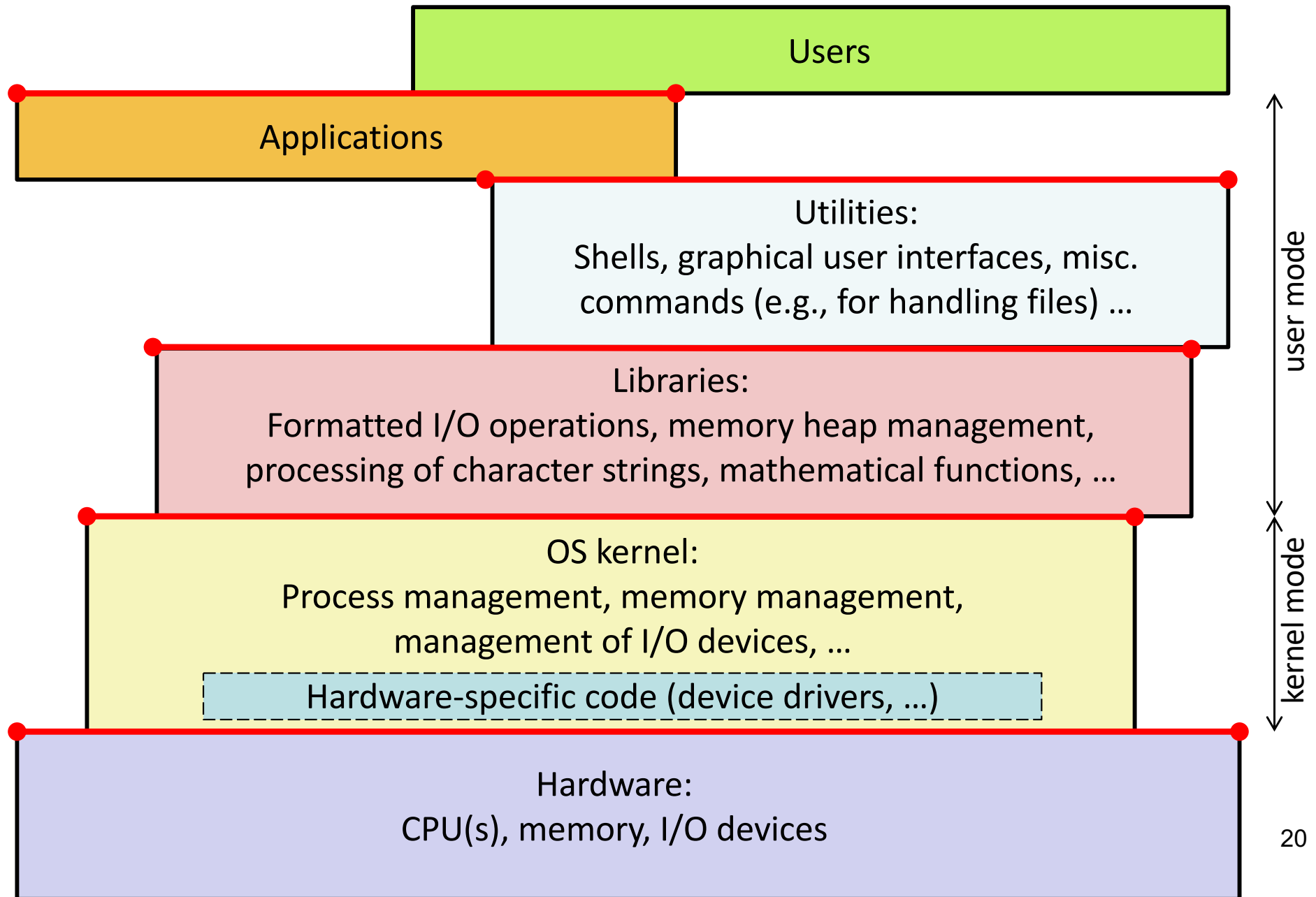
- Realization that the code of the OS plays a central role
- **A user/application should not be able to make the whole system fail or to perform unauthorized operations**
  - E.g., issue arbitrary write requests to a storage device
- Idea: Modification of the OS interface
  - Old interface: provide applications with library procedures allowing direct access to critical operations
  - New interface: **force application to delegate critical operations, using a hardware mechanism** that transfers control to a more privileged execution mode
    - Such an interface is called a “system call” or “syscall” (more details later)

# Some History

## (3) Multiprogramming / Multitasking

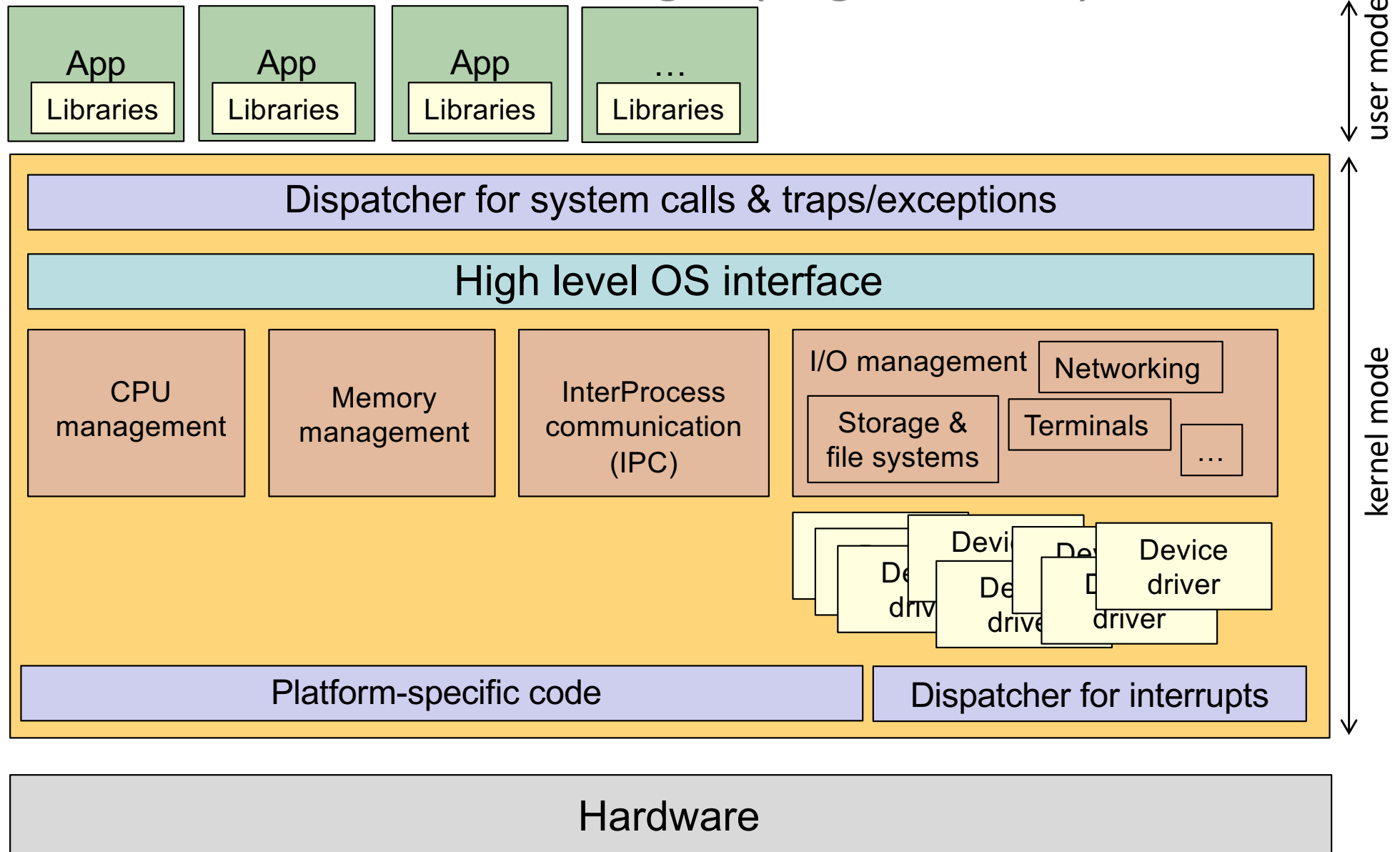
- Idea: **improve machine resource utilization by running several programs concurrently**
  - When a program blocks (e.g., waiting for input from the disk / the network / the user), run another program
- **Problems:** what can an ill-behaved application do ?
  - Never relinquish the CPU (infinite loop)
  - Access the memory of another application
- **The OS provides mechanisms to address these problems**
  - Preemption: take CPU away from a looping application
  - Memory protection: prevent an application from accessing another application's memory

# Typical structure of an operating system

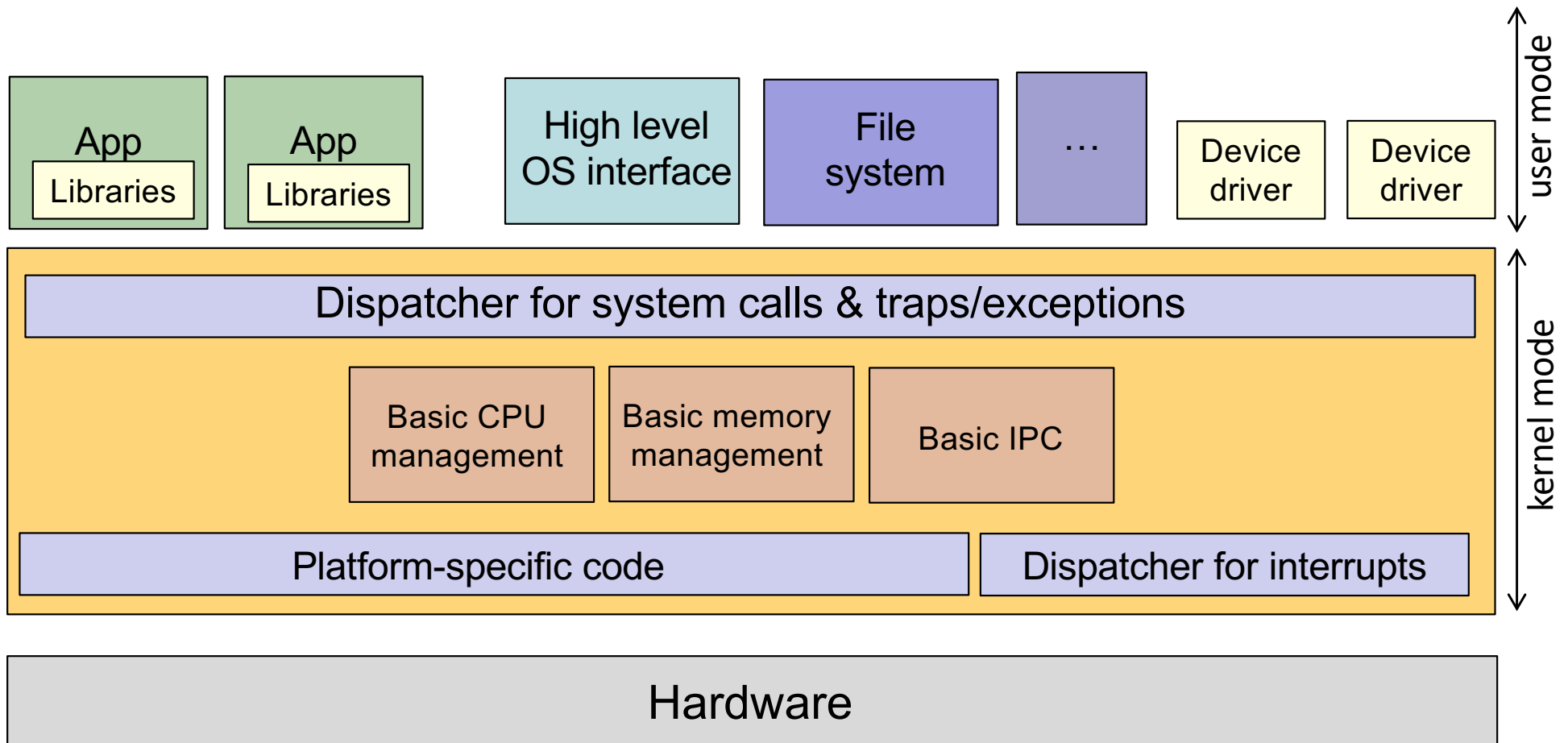




# Monolithic kernel design (e.g., Linux)



# Microkernel design (e.g., L4)



# Outline

- What is an operating system?
- Some history
- **Abstractions: processes and address spaces**
- Protection and resource management

# A key OS abstraction: the Process

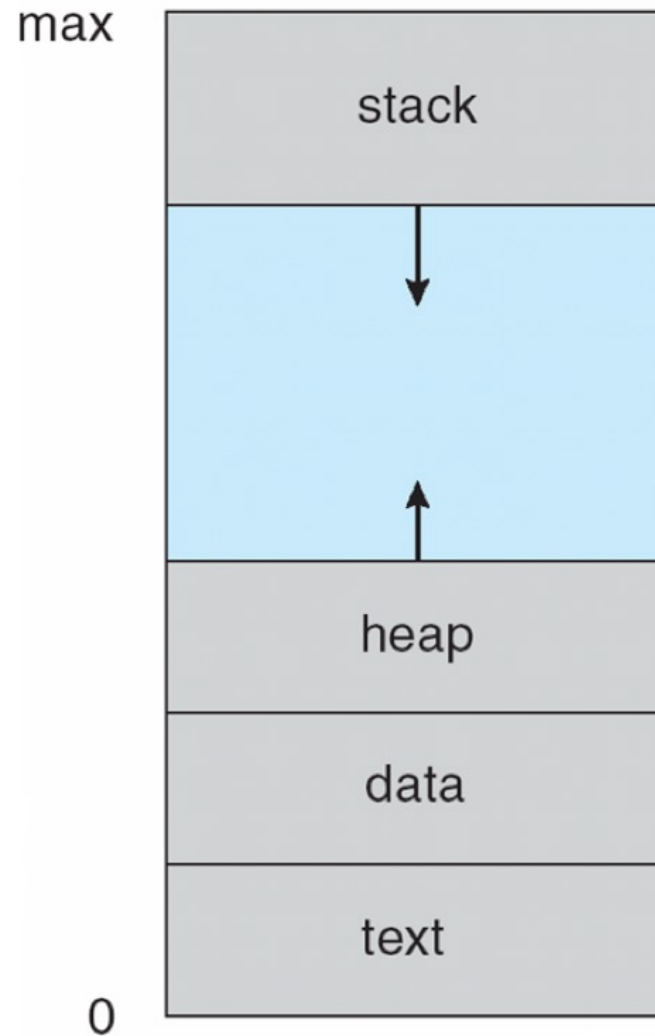
- **A *process* is an abstraction corresponding to a running instance of a program**
- Its main role consists in **virtualizing the CPU**
  - Although there are just a few physical CPUs (or even just one), the OS can provide the illusion of a nearly-endless supply of logical CPUs (one per process)
  - Its also allows the OS to capture the state and control the execution of a running program, which are key mechanisms for resource management

# A key OS abstraction: the Process

- A process mainly consists in:
  - An **execution context** (a.k.a. an *execution flow*, or a *control flow*):
    - A current machine state: a set of current values for the CPU registers, including the program counter (PC) and the stack pointer (SP)
    - An execution stack
  - A **memory space** (a.k.a. an *address space*)
  - A **logical state** (is it currently running? If not, why?)
  - Some other information, required by the OS

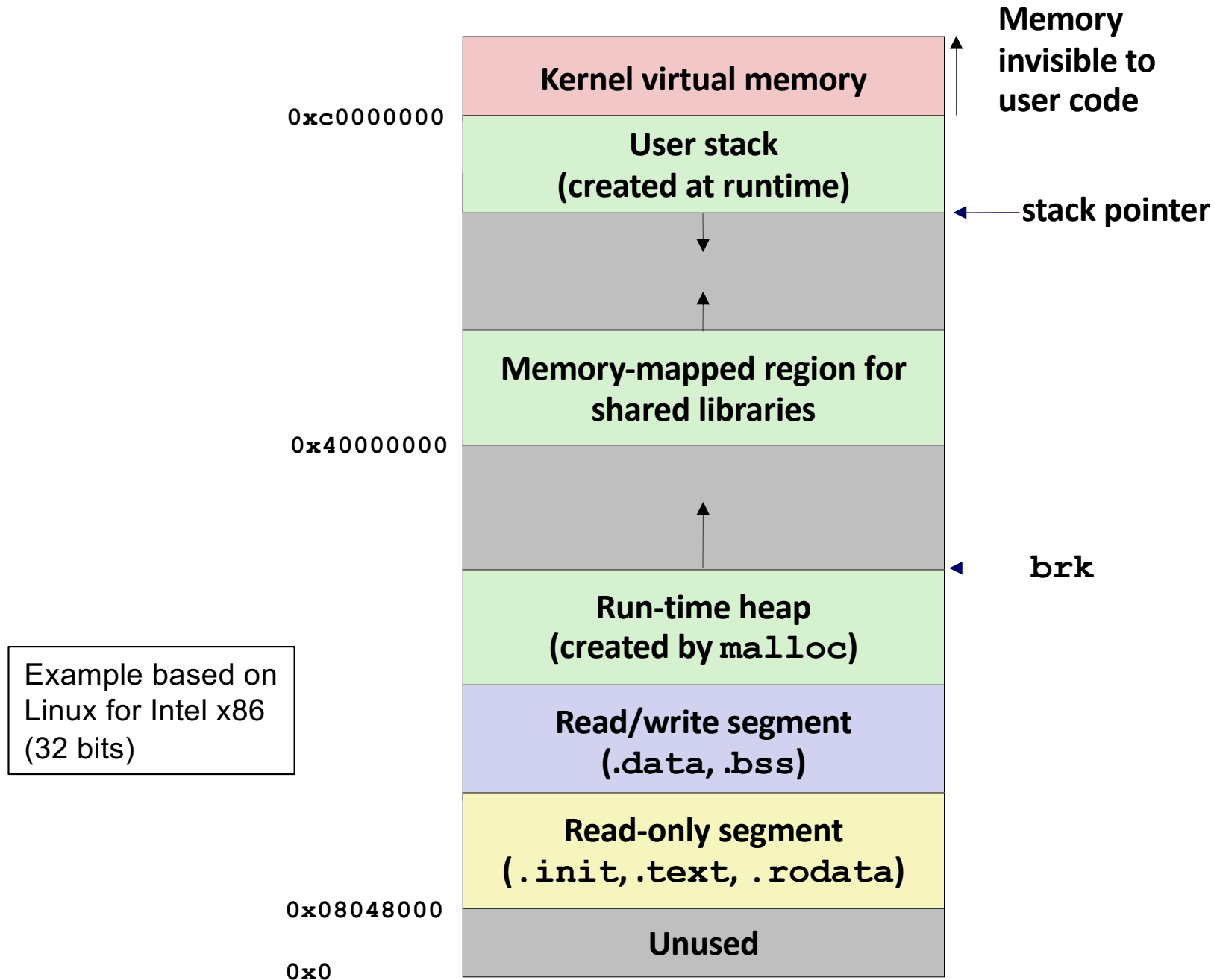
# Process address space

## A simplified view



# Process address space

## A more detailed view



# Outline

- What is an operating system?
- Some history
- Abstractions : processes and address spaces
- **Protection and resource management**



# Some key techniques for protection

- Overall goal: **prevent bad processes from impacting the OS or other processes**
- **Preemption**
  - **Give a resource to a process and take it away if needed** for something else
  - Example: CPU preemption
- **Interposition**
  - **Place OS between application and resources** (e.g., an I/O device, or a piece of information stored in memory)
  - OS tracks the resources that the application is allowed to use
  - On every access request, check that the access is legal
  - Example: System calls

# Some key techniques for protection (continued)

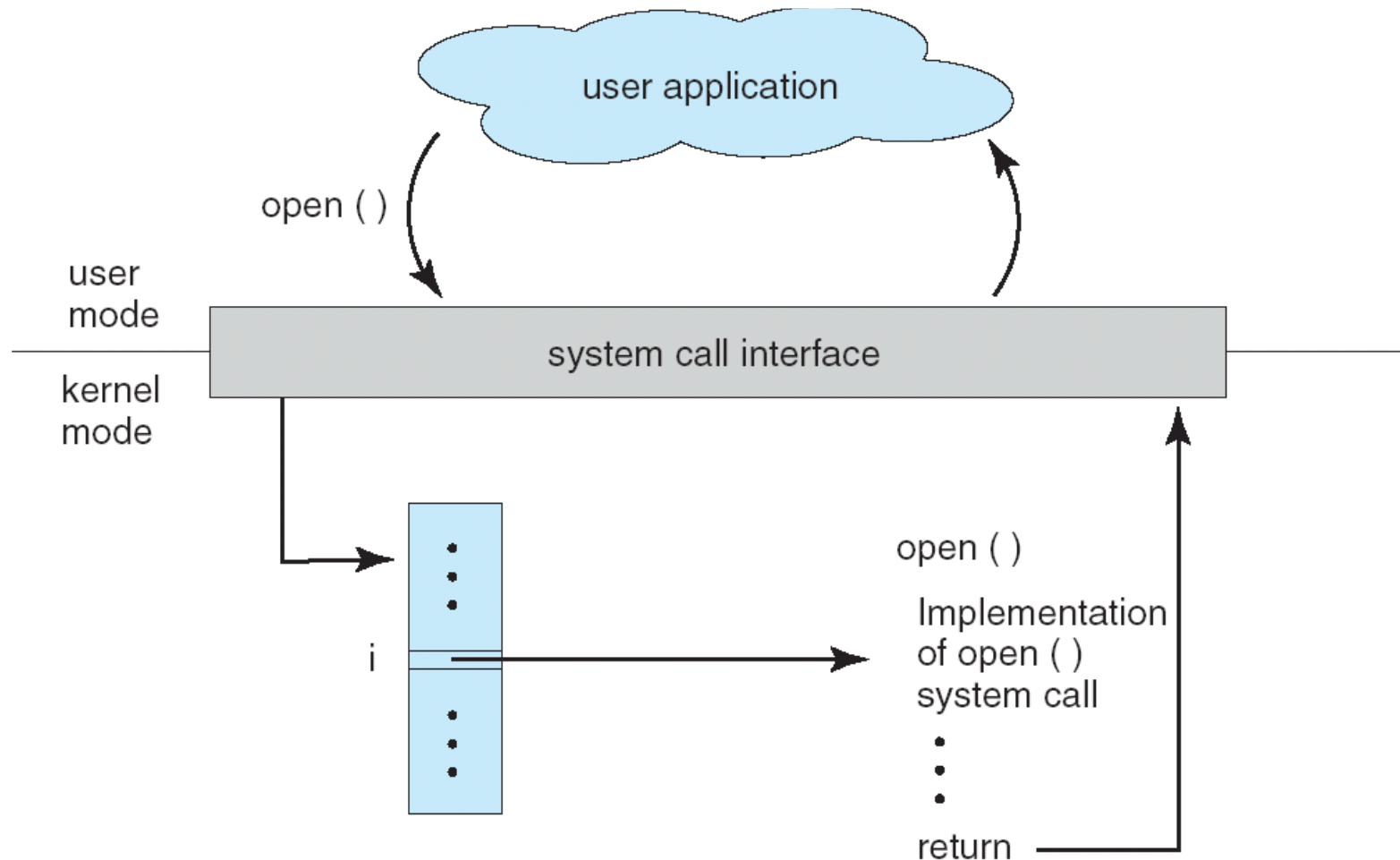
- CPU execution modes
  - CPUs provide **2 execution modes**:
    - **Privileged** (a.k.a. supervisor mode, or kernel-level mode)
    - **Unprivileged** (a.k.a. user mode, or user-level mode)
  - **OS kernel code** runs in **privileged mode**
  - **Application code** runs in **unprivileged mode**
  - Protection-related code (resp. data) must only be executed (resp. accessed) in privileged mode
    - Enforced by hardware (details later)
    - A system call is the only way to switch from unprivileged mode to privileged mode

# System calls

- Applications (i.e., user-level code) can **invoke kernel services** through the system call mechanism
  - **Using a special hardware instruction** that triggers a trap into kernel-mode
  - ... and transfers control to a trap handler
  - ... which dispatches to one of a few hundred syscall handlers

# System calls (continued)

- Illustration with the `open` system call (to open a file)

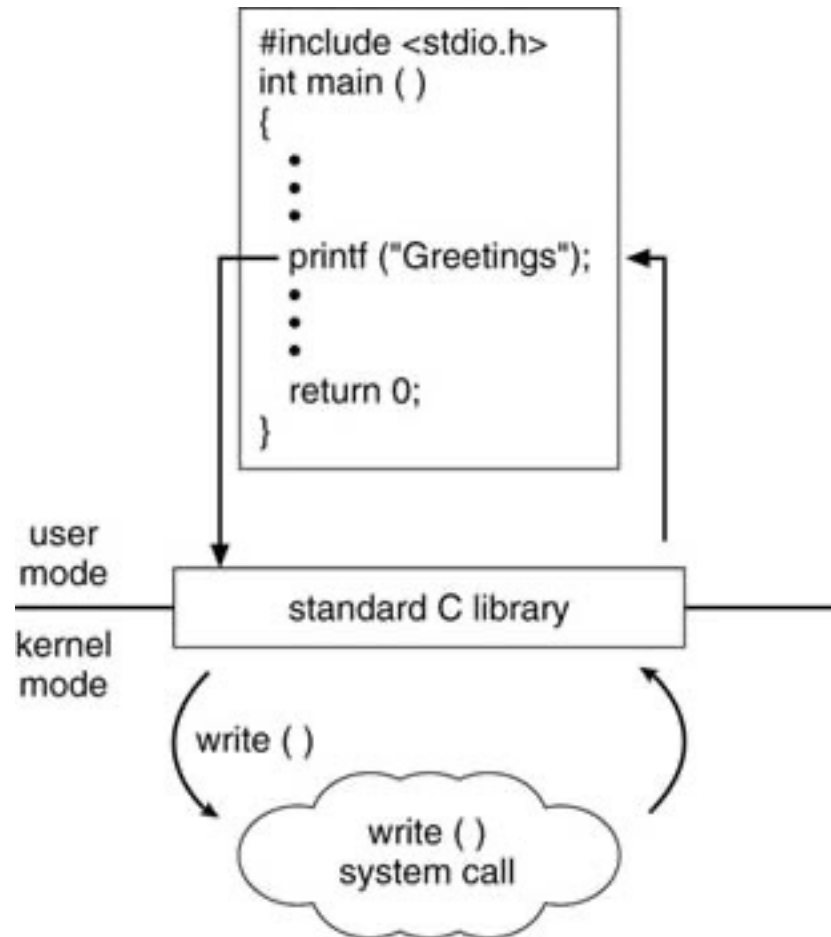


# System calls (continued)

- Goal: **perform things that an application is not allowed to do in unprivileged mode**
  - Like a library call, but into more privileged code
- The kernel supplies a **well-defined system-call interface**
  - Applications set up syscall arguments and trap to kernel
  - Kernel checks if operation is allowed, performs operation and returns results (transfers control back to application)
  - **Many higher-level library functions are built on the syscall interface**
    - Example (Unix) : functions such as **printf** and **scanf** are implemented as user-level library code that calls the kernel using system calls such as **read** and **write**

# System call example

- The standard C library (libc) is implemented in terms of syscalls
  - **printf** (in libc) has same privilege as application
  - **printf** calls **write**, which can access low-level resources such as the console/screen and files



# CPU preemption

- **Protection mechanism to prevent a process from monopolizing the CPU**
  - Allows the kernel to take back control of the CPU after a maximum time interval
  - Relies on the processor interrupt mechanism and on a timer device
- **The kernel programs the timer to send periodic interrupts (e.g., every 10 ms)**
  - Device configuration is only allowed in privileged mode
  - User code cannot re-program the timer

# CPU preemption (continued)

- The kernel configures the processor to set up a **timer interrupt handler**
  - This handler is a piece of code provided by the kernel, and runs in privileged mode
  - In this way, **each periodic timer interrupt will trigger the execution of some kernel-defined code**
  - This kernel code can decide to keep the current process running or to give the CPU to another one
  - Note : interrupt handlers cannot be defined/modified by user-level code
    - Thus, there is no way for user code to hijack the interrupt handler
- **Result: a process cannot monopolize the CPU with an infinite loop**
  - At worst, it may get  $1/N$  of CPU time if there are  $N$  CPU-hungry processes



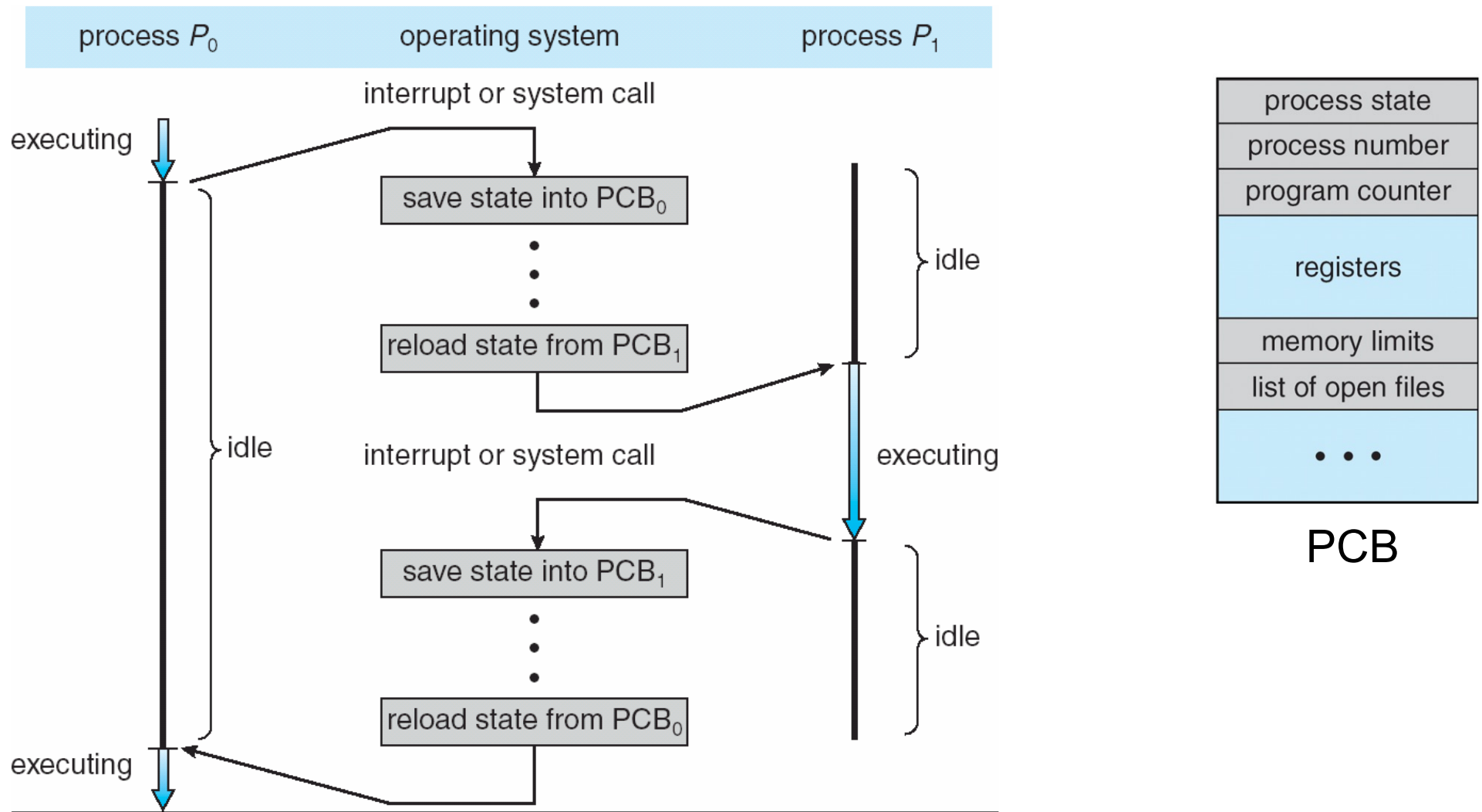
# CPU scheduling

- **The *scheduler* is a component of the OS, in charge of deciding which process should run on the CPU** (1 decision per CPU)
- When is the scheduler invoked?
  - **Periodically, for each timer interrupt**
  - **Punctually, in reaction to some syscalls:**
    - Process termination (exit)
    - Process explicitly releasing the CPU (yield, sleep, ...)
    - Process requesting a blocking action
    - Creation of a new process with a higher priority
    - ...
  - **Punctually, in reaction to some interrupts**
    - E.g., a device notification for available data

# CPU scheduling (continued)

- **What does the scheduler do upon invocation?**
  - **Make decision on the process P2 that should obtain the CPU**, based on:
    - The list of processes that are ready to run
    - ... and a given scheduling policy
  - **Save execution context of “outgoing” process P1**
    - (Except if this process is terminated)
    - This allows resuming the execution of P1 later on
  - **Inject /restore the execution context of P2 on the CPU**
- This sequence of steps is called a “**context switch**”
  - Note that, just after the switch, P2 runs in kernel mode and must eventually switch back to user mode. This will happen via a return-from-interrupt or a return-from-syscall instruction.

# Context switch



# Context switch (continued)

A simplified code example (taken from “xv6”, a pedagogical mini-OS developed by MIT – [this version is for the RISC-V 64-bit processor](#))

```
struct context {
    uint64 ra;
    uint64 sp;
    // callee-saved
    uint64 s0;
    uint64 s1;
    uint64 s2;
    uint64 s3;
    uint64 s4;
    uint64 s5;
    uint64 s6;
    uint64 s7;
    uint64 s8;
    uint64 s9;
    uint64 s10;
    uint64 s11;
};
```

```
# void swtch(struct context *old, struct context *new);
# Save current register context in old
# and then load register context from new.
```

```
.globl swtch
```

```
swtch:
```

```
    # Save old registers
```

```
    sd ra, 0(a0)
```

```
    sd sp, 8(a0)
```

```
    sd s0, 16(a0)
```

```
    sd s1, 24(a0)
```

```
    sd s2, 32(a0)
```

```
    sd s3, 40(a0)
```

```
    sd s4, 48(a0)
```

```
    sd s5, 56(a0)
```

```
    sd s6, 64(a0)
```

```
    sd s7, 72(a0)
```

```
    sd s8, 80(a0)
```

```
    sd s9, 88(a0)
```

```
    sd s10, 96(a0)
```

```
    sd s11, 104(a0)
```

```
    # Load new registers
```

```
    ld ra, 0(a1)
```

```
    ld sp, 8(a1)
```

```
    ld s0, 16(a1)
```

```
    ld s1, 24(a1)
```

```
    ld s2, 32(a1)
```

```
    ld s3, 40(a1)
```

```
    ld s4, 48(a1)
```

```
    ld s5, 56(a1)
```

```
    ld s6, 64(a1)
```

```
    ld s7, 72(a1)
```

```
    ld s8, 80(a1)
```

```
    ld s9, 88(a1)
```

```
    ld s10, 96(a1)
```

```
    ld s11, 104(a1)
```

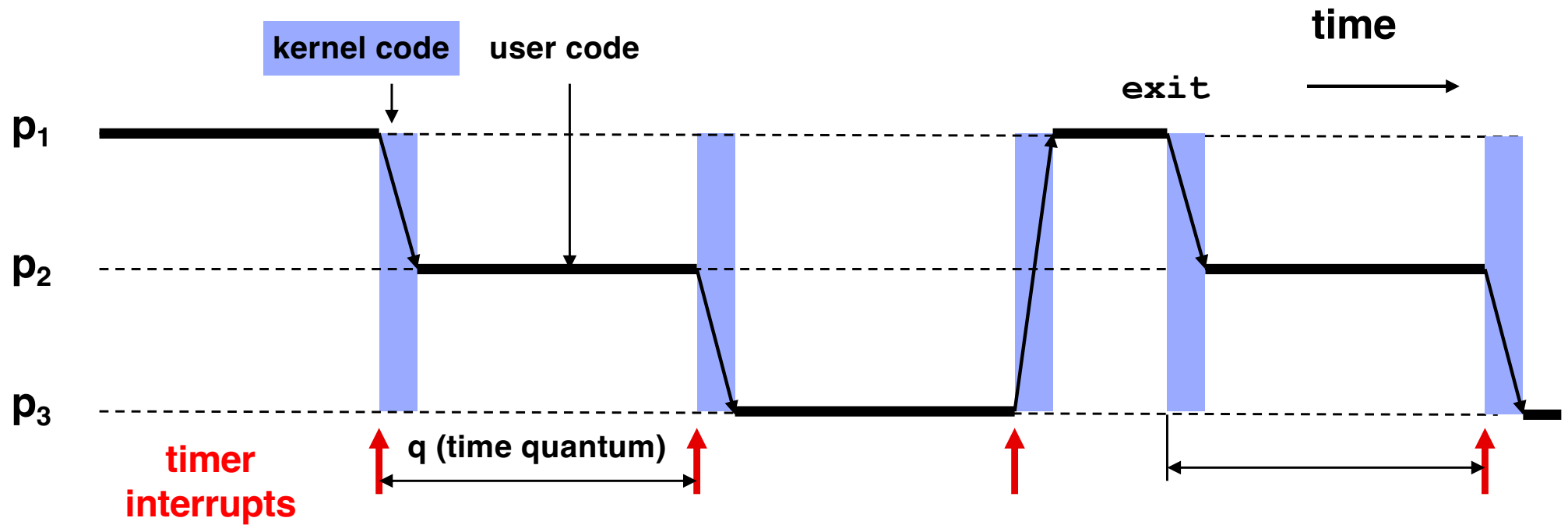
```
    # Finally return into new ctxt
```

```
    ret
```

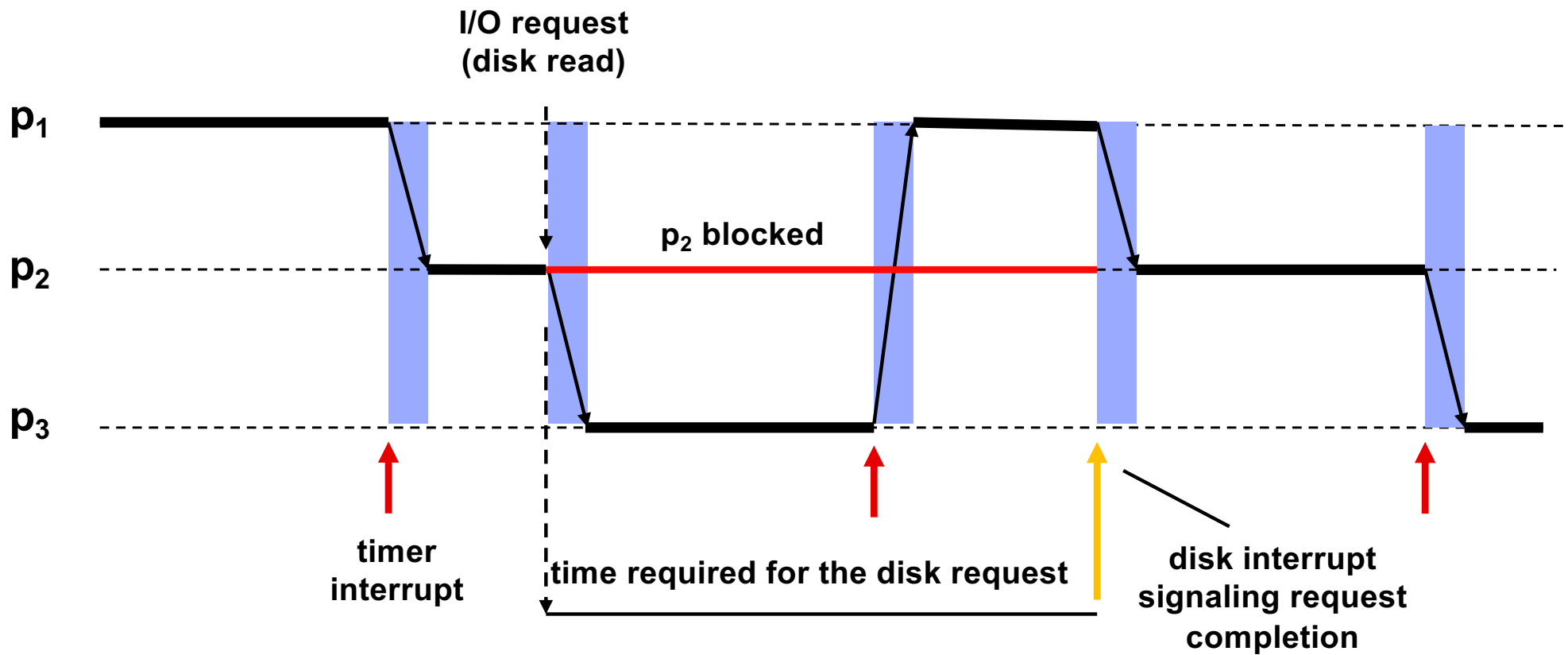
# Context switch (continued)

- Notes:
  - Implementation details are very machine (processor) dependent, but the general principle is the same
  - A context switch has **a non-negligible cost** and should not happen too often
  - **Warning: Do not confuse**
    - **Context-switch** (transition between two execution contexts)
    - **Mode switch** (transition between user and kernel mode, in the same execution context)

# CPU scheduling examples



# CPU scheduling examples (continued)



# Memory virtualization and protection

- **The OS must protect the memory space of a process from the actions of other processes**
- Definitions
  - **Address space**: all memory locations that a program can name
  - **Virtual address**: an address in a process address space
  - **Physical address**: an address in real memory
  - **Address translation** : map virtual address to physical address
- **A translation is performed for each executed instruction that issues a memory access**
  - Modern CPUs do this in hardware for speed
- Idea: if you cannot name it, you cannot touch it
  - Ensure that the translations of a process do not include memory areas of other processes



# Memory virtualization and protection (continued)

- **CPU allows kernel-only virtual addresses**
  - The kernel is typically part of all address spaces, e.g., to handle a system call in the same address space
  - But the OS must ensure that applications cannot touch kernel memory
- **CPU allows disabled virtual addresses**
  - Helps catching and halting buggy program that makes wild accesses
  - Makes virtual memory seem bigger than physical (e.g., bring a page in from disk only when accessed)
- **CPU allows read-only virtual addresses**
  - E.g., allows sharing of code pages between processes
- **CPU allows disabling execution of virtual addresses**
  - Makes certain (code injection) security attacks harder

# Summary

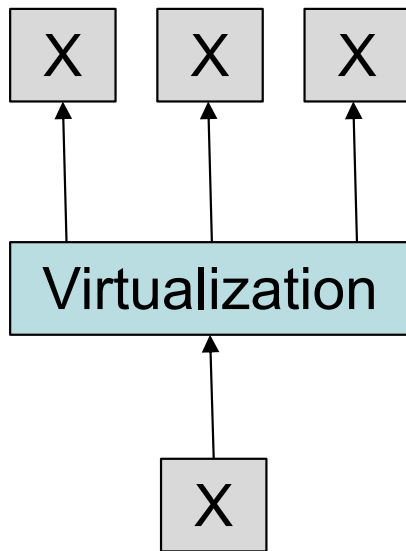
- The main roles of an OS are **virtualization** and **resource management**
- Protection is a fundamental concern
- Some **key abstractions**
  - Processes
  - Virtual address spaces
- Some **key mechanisms (hardware-assisted)**
  - Privileged/unprivileged execution modes
  - System calls and traps
  - CPU preemption (relying on processor interrupts)
  - Memory translation (implementation will be studied in next lectures)

# Appendix 1: [Optional] Virtualization

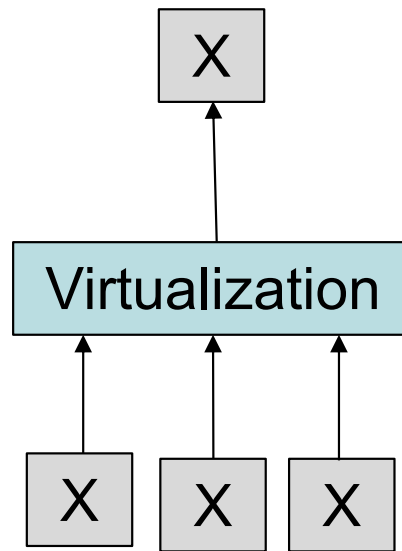
# Main techniques for virtualization (1/5)

- In order to virtualize the resources of a machine, operating systems rely on **a combination of 3 main techniques**:
  - Multiplexing (in space and/or in time)
  - Aggregation
  - Emulation
- Note: these techniques are sometimes also used within some hardware devices.

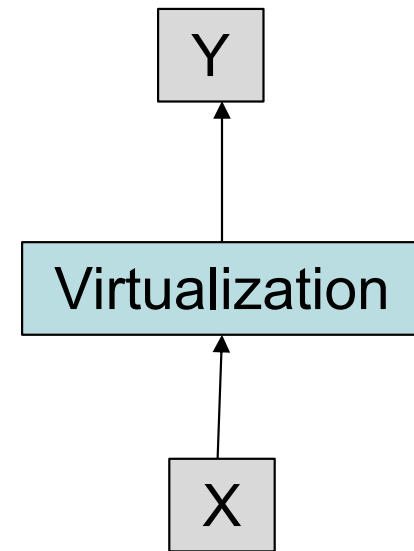
# Main techniques for virtualization (2/5)



Multiplexing



Aggregation



Emulation

# Main techniques for virtualization (3/5)

- Multiplexing:
  - Exposes a resource among multiple virtual entities
  - Two types of multiplexing: in space and in time
  - Examples:
    - CPUs (in time)
    - Memory (in space and possibly also in time when using swapping)
    - I/O devices (in time, and also in space for storage devices)

# Main techniques for virtualization (4/5)

- Aggregation:
  - The opposite of multiplexing
  - Takes multiple resources and makes them appear as a single abstraction
  - Examples:
    - Memory controller with several DIMMS (hardware)
    - RAID (hardware or software)

# Main techniques for virtualization (5/5)

- Emulation
  - Expose (using a level of indirection in software ) a virtual resource that is not provided by the underlying machine
  - Examples
    - Sockets and files provide higher-level abstractions above hardware devices
    - Binary translation (compatibility layer)
      - A CPU emulator can run programs compiled for a given processor family X (e.g., Intel) on another processor family Y (e.g., ARM)



## Appendix 2: **[Optional]** **Observing system & library calls**

# Tracing library & system calls

- It is possible to obtain a trace the calls performed by a process via specific tools.
- This is useful for many different purposes: debugging, performance troubleshooting, reverse engineering, understanding complex applications ...
- In Unix/Linux systems:
  - The **strace** utility allows tracing **system calls**
  - The **ltrace** utility (or a variant like **latrace** or **uftrace**) allows tracing **library calls**

# A simple tracing example

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t mypid;
    printf("Hello\n");
    mypid = getpid();
    printf("My pid is: %ld\n",
        (long)mypid);
}
```

```
$ gcc -Wall -o test.run test.c
```

```
$ ltrace ./test.run
```

```
__libc_start_main(0xaaaabec50814, 1, 0xfffffe4670ff8, 0
```

```
<unfinished ...>
```

```
puts("Hello"Hello) = 6
```

```
getpid() = 21198
```

```
printf("My pid is: %ld\n", 21198My pid is: 21198) = 17
```

```
__cxa_finalize(0xaaaabec61008, 0xaaaabec507c0, 0x10d88, 1) = 1
```

```
+++ exited (status 0) +++
```

# A simple tracing example (continued)

```
$ strace ./test.run

execve("./test.run", ["./test.run"], 0xffffd9332c50 /* 56 vars
*/) = 0
[ ... Many initialization system calls omitted for simplification]
write(1, "Hello\n", 6Hello)                = 6
getpid()                                   = 21391
write(1, "My pid is: 21391\n", 17My pid is: 21391)    = 17

exit_group(0)                               = ?
+++ exited with 0 +++
```

# A simple tracing example (continued)

```
$ strace -c ./test.run
```

```
Hello
```

```
My pid is: 21506
```

% time	seconds	usecs/call	calls	errors	syscall
23,50	0,000329	329	1		execve
15,71	0,000220	36	6		mmap
9,71	0,000136	34	4		mprotect
8,14	0,000114	38	3		newfstatat
7,14	0,000100	33	3		munmap
6,57	0,000092	30	3		brk
5,29	0,000074	37	2		openat
4,71	0,000066	33	2		write
3,07	0,000043	43	1	1	faccessat
2,93	0,000041	20	2		close
2,29	0,000032	32	1		set_tid_address
2,07	0,000029	29	1		read
1,93	0,000027	27	1		rseq
1,86	0,000026	26	1		getrandom
1,71	0,000024	24	1		set_robust_list
1,71	0,000024	24	1		prlimit64
1,64	0,000023	23	1		getpid
100,00	0,001400	41	34	1	total