CSC369 Lecture 2

Larry Zhang, September 21, 2015

Volunteer note-taker needed by accessibility service

see announcement on Piazza for details

Change to office hour
to resolve conflict with CSC373 lecture

Monday change from 11am-12pm to 3pm-4pm

Assignment 1 will be out tomorrow!

➔ Due: October 14, 9:59pm

➔ Start early, and ask for clarification.

Assignment 1

- ➔ You will do some serious kernel programming.
- ➔ You will add to the kernel your own system call, which can be used to "hijack" other system calls.
- ➔ General tips:
  - ◆ A significant amount of time will be spent in reading / understanding the handout / starter code, which is fine. Don't start writing code prematurely.
  - ◆ Read the comments in starter code.
  - ◆ Learn how to search for documentation of existing kernel functions.
- ➔ Working in groups of up to 2 students.
  - ◆ Don't be a leech!
  - ◆ Don't allow your partner to become a leech!
  - ◆ This is your chance to find out whether you belong to this course or not. Feeding off other people may give your the wrong impression that things are going well, which will bite you badly in tests and exams -- that's almost the only way to fail this course.

Review last week

- What is a process:
  - abstraction of execution, virtualization of CPU
- How processes are stored and represented in OS
  - PCB
  - State queues
- How to create a new process
  - fork()
  - exec()
- How to terminate a process
  - zombie state

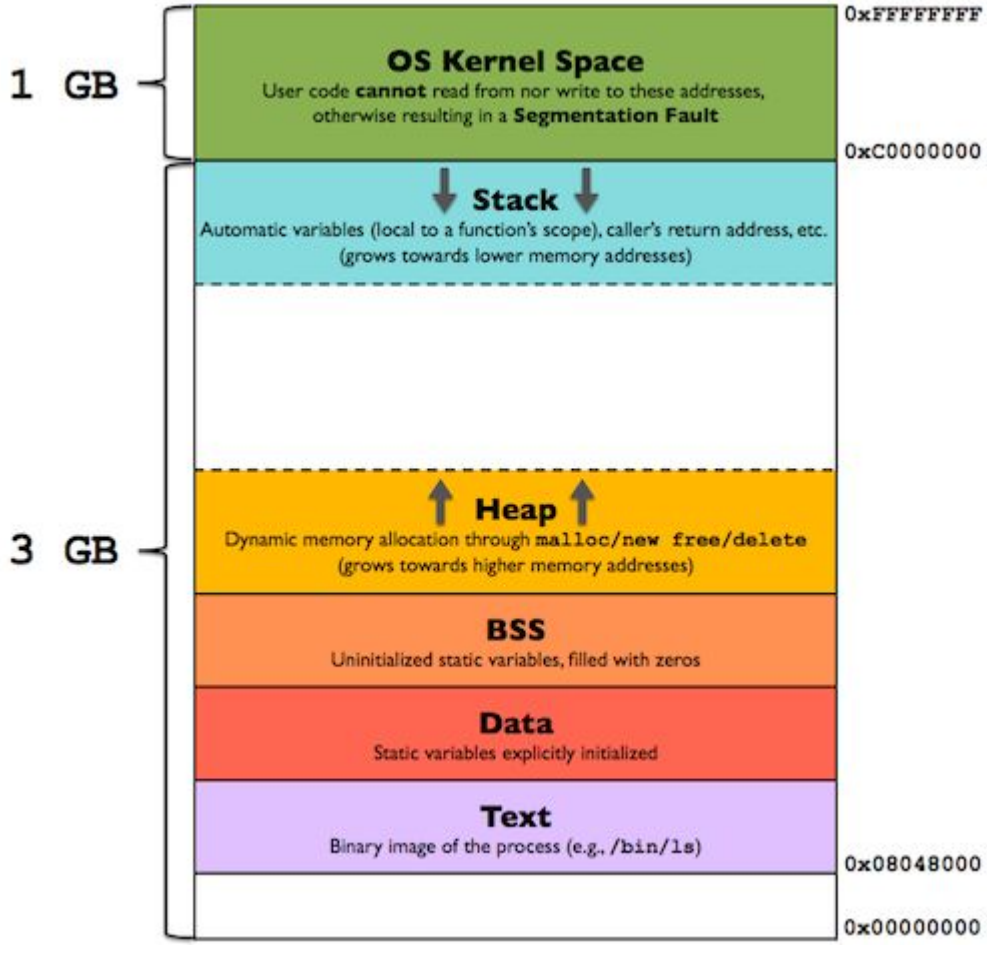
What is the illusion created by virtualizing CPU?

What is the illusion created by virtualizing memory?

Where is user stack?

Where is kernel stack?

Which area does PC point to?



What info does a PCB keep?

- Process state (ready, running, blocked, ...)
- The **address space**
- The **code** of the program
- The **data** of the program
- Execution **stack.**
- The **program counter (PC)**
- A set of general-purpose **registers** with current values.
- A set of operating system **resources**
  - open files, network connections, signals, etc.
- CPU scheduling info: process **priority**
- Each process is identified by its **process ID (PID)**

Today’s outline

➔ Processes cont.

➔ System calls

Exercise

Function call procedure

(not the only answer,
not architecture-specific)

```
int pinkbunny(int x, int y) {
    int i = 10;
    return x + y + i;
}

int main() {
    int r = 2;
    int q = 3;
    int result = pinkbunny(r, q);
    printf("result is %d\n", result);
    return 0;
}
```

2.
pop args from stack

3.
store return value in register (e.g., EAX)
jump to return address (value in $ra)

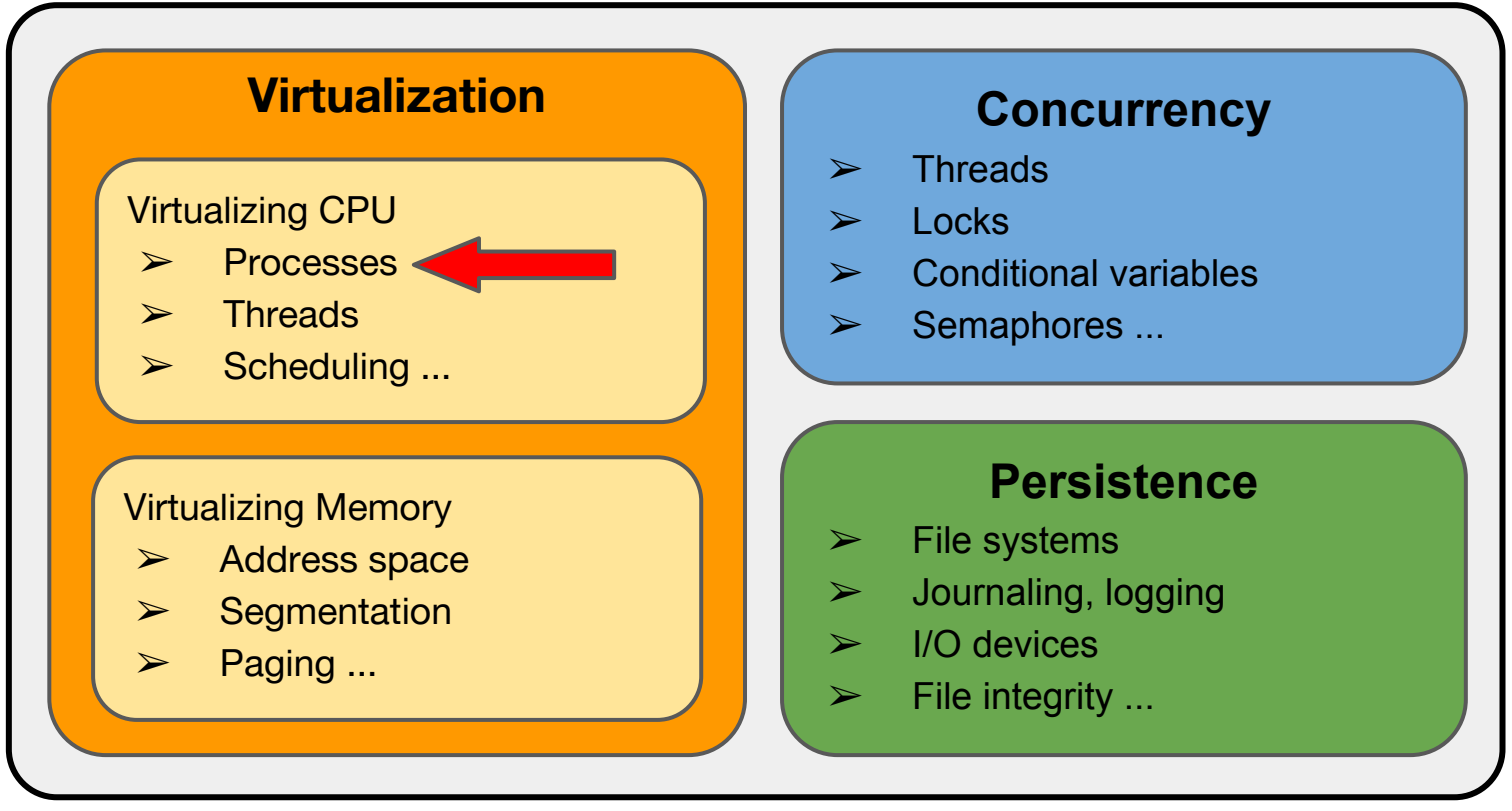
1.
push registers to stack,
push args to stack,
push return address to register (e.g., $ra)
jump to pinkbunny

4.
restore registers from stack
assign result with value in EAX
continue the rest ...

More details about the mechanism of process execution

“Limited Direct Execution”

We are here


Virtualization
Virtualizing CPU
Processes
Threads
Scheduling ...
Virtualizing Memory
Address space
Segmentation
Paging ...
Concurrency
Threads
Locks
Conditional variables
Semaphores ...
Persistence
File systems
Journaling, logging
I/O devices
File integrity ...

So, it is about virtualizing CPU

- ➔ A computer with only one CPU can have multiple processes (**seemingly**) running at the same time.
- ➔ Different processes need to share the physical CPU
  - ◆ In particular, they do **time sharing**, to achieve CPU virtualization.

Two design goals of CPU virtualization mechanisms

- ➔ Performance
  - ◆ We want the process to be itself and run as fast as possible, without having to interact with OS frequently.
- ➔ Control
  - ◆ We want to avoid one process running forever and taking over the machine, or accessing what it should not access

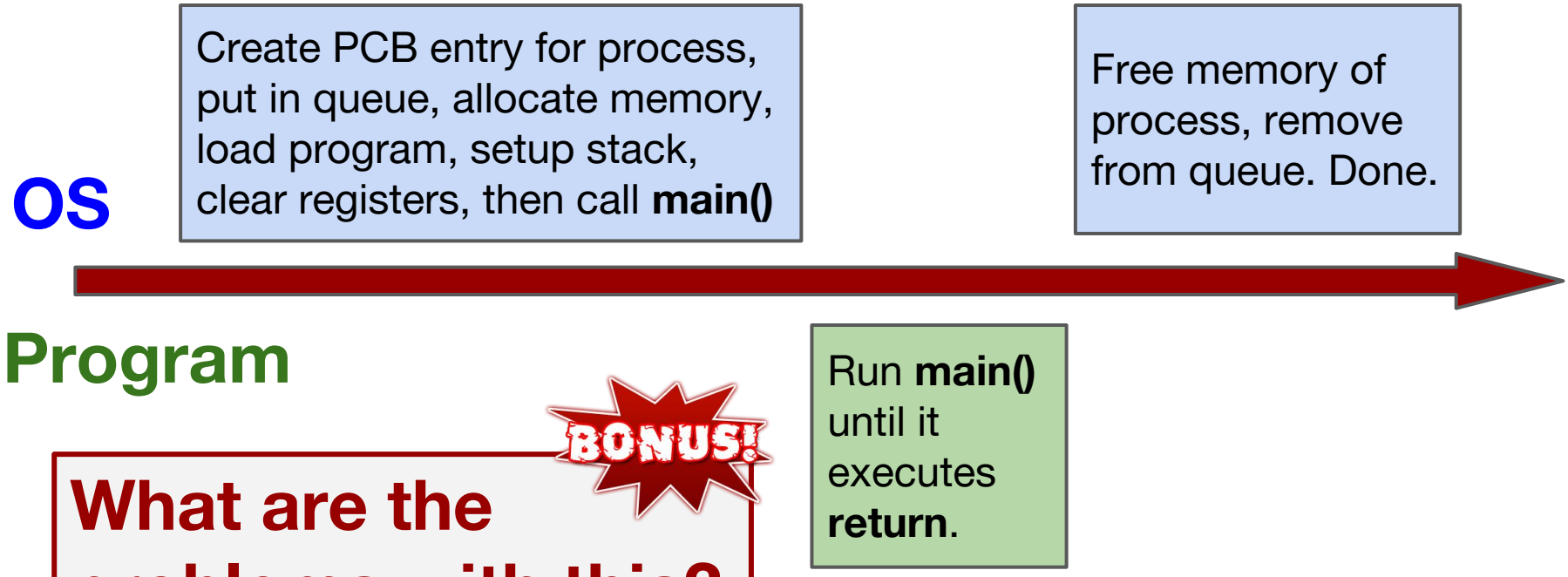
These two goals look contradictory!

The Crux:
efficiently virtualize CPU with **control**

First, focus on "**efficient**" and see what we can do

➔ The most efficient way to execute a process:

- **Direct Execution**



The problems with **Direct Execution**

➔ Once the program starts to run, the OS becomes a complete outsider -- it doesn't have any **control** over the running program

a. The program can access anything it wants to, including doing **restricted operations**.
b. The program may never want to **switch** to a different process, which fails the idea of **time sharing**.

Let's solve these two problem one by one.

Solve Problem #1
Restricted Operations

Restricted (Privileged) Operations

- ➔ The process should be able to perform restricted operations, such as disk I/O, open network connections, etc.
- ➔ But we should **not** give the process **complete control** of the system.
- ➔ **Solution: hardware support -- processor modes**
  - ◆ **user mode and kernel mode**

User Mode and Kernel Mode

- ➔ A **mode bit** added to **hardware** to support distinguishing between user mode and kernel mode
- ➔ Designate some instructions as **privileged instructions** that cannot be run in user mode (only in kernel mode)
  - ◆ a user-mode process trying to perform privileged instructions will raise a *protection fault*, then be killed.
- ➔ So how can a user-mode process go into to kernel mode and perform the restricted operations?
  - ◆ Make a **system call**

82	sys_rename	const char *oldname	const char *newname
83	sys_mkdir	const char *pathname	int mode
84	sys_rmdir	const char *pathname	
85	sys_creat	const char *pathname	int mode
86	sys_link	const char *oldname	const char *newname

System Calls

- A **small set** of APIs for restricted operations
  - Linux x86_64 has ~323 system calls, numbered from 0~322
  - OS uses a sys_call_table to keep the syscall handlers (indexed by syscall number)

  http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64

- Now we can only perform ~300 different kinds of restricted operations, instead of doing anything we want.
  - in other words, the process is able to perform **restricted operations**, but does not get **complete control** over the system.

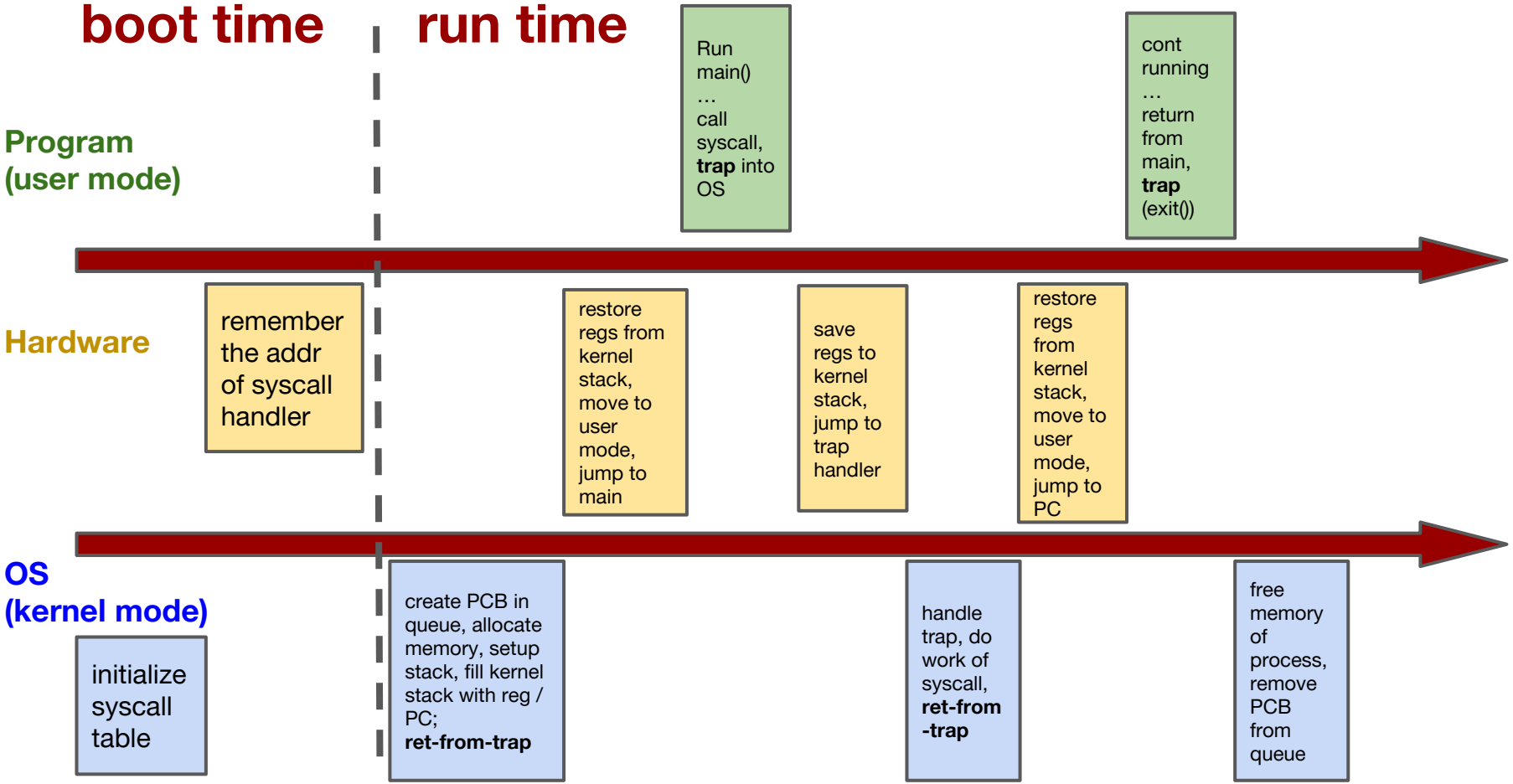
Problem #1 solved!

More detail: How a system call happens

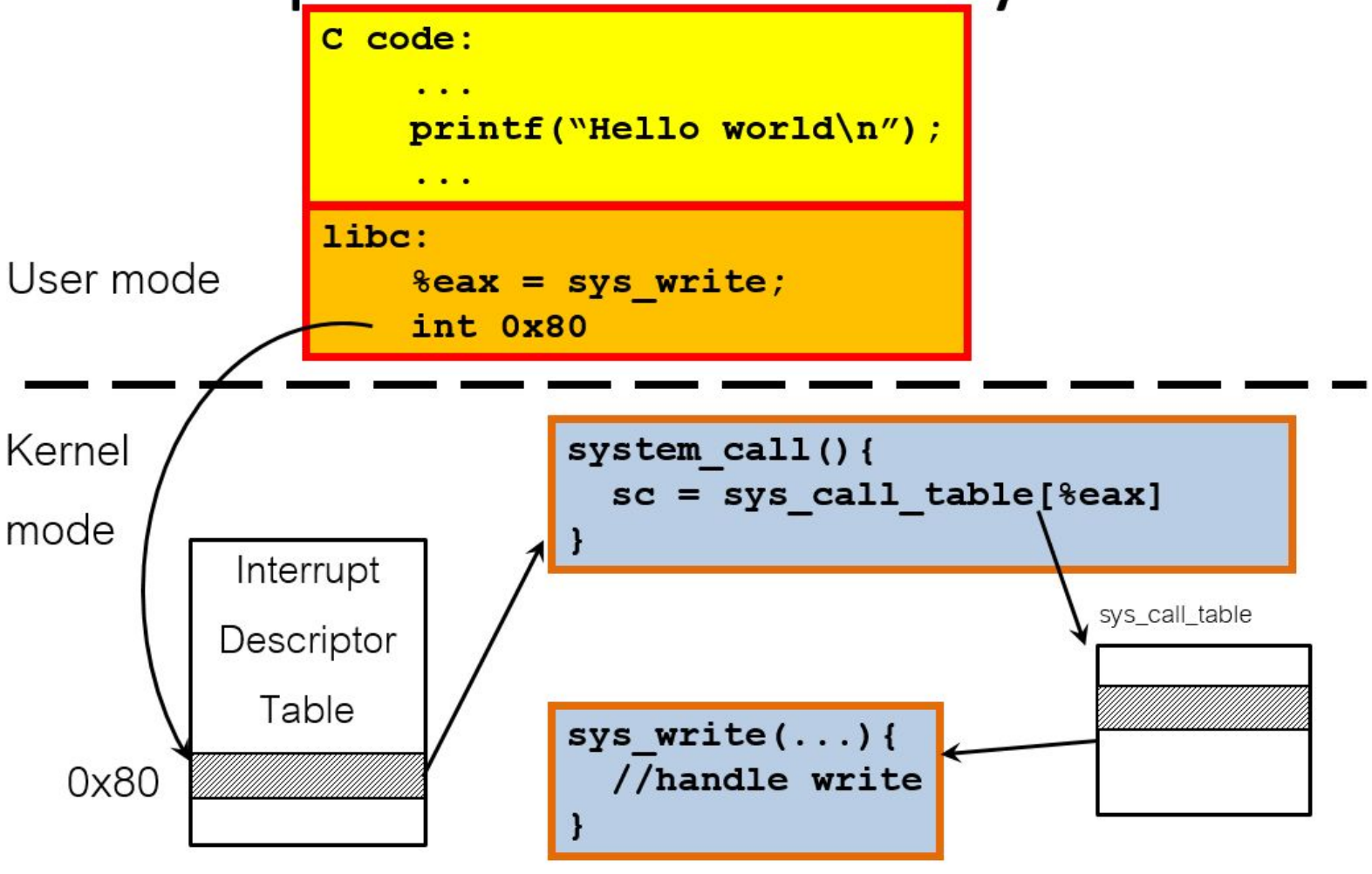
- ➔ Think of a user process that makes some system call.
- ➔ Need to switch from user mode to kernel mode, do the privileged operation, then switch back.
- ➔ Need **hardware support** for doing this switch
  - ◆ What is that assembly instruction that does this?

- ➔ **trap**: go from user mode to kernel mode
- ➔ **return-from-trap**: go back from kernel mode to user mode

How a system call happens (detailed diagram)



Example: Linux “write” system call



System calls ***vs*** normal C function calls


```
asmlinkage long sys_mkdir(const char __user *pathname, umode_t mode);

long greatest_common_divisor(long x, long y);
```

What's the difference?

- ➔ System calls appear to be just like function calls, but …
- ➔ System calls have **trap instruction** in them (the main difference).
- ➔ System calls have an extra level of indirection (trapping to kernel mode and returning user mode), whereas functional call stays in userspace.
- ➔ System calls perform restricted operations, whereas function calls do not.
- ➔ System calls have bigger overhead and are slower than equivalent function calls.
- ➔ System calls use kernel stack, function calls use user stack

detour: some implementation details

Linux system call naming

Example: the **`write()`** system call

➔ User space definition (in libc):

```
long write(unsigned int fd, const char* buf, size_t count);
```

➔ Kernel space definition:

```
asmlinkage long sys_write(unsigned int fd, const char __user
*buf, size_t count);
```

Always starts with “sys_”

The user space function will eventually call the kernel space function

Linux system call dispatching

➔ Can invoke any system call using the **`syscall()`** function

```
syscall(syscall_no, arg1, arg2, arg3, ..)
```

For example:

```
const char msg[] = "Hello World!";

syscall(4, STDOUT_FILENO, msg, sizeof(msg)-1);
```

is equivalent to:

```
write(STDOUT_FILENO, msg, sizeof(msg)-1);
```

`write()` has syscall number 4

Linux system call dispatching

How does **`syscall()`** find the right routine to execute given **`syscall_no`**?

Could do this:

```
if (syscall_no == 0) {
    sys_setup(regs);
}
...
else if (syscall_no == 4) {
    sys_write(regs);
}
... 300 more else if
```

Lame

Use a **system call table**, an **array** of ~300 **function pointers**, then essentially you can do it like

```
sys_call_table[syscall_no](regs);
```

How are these arguments passed?

Passing system call parameters to `syscall()`

- The first parameter is always the syscall number
  - Stored in eax
- Can pass up to 6 parameters:
  - ebx, ecx, edx, esi, edi, ebp
- If more than 6 parameters are needed, package the rest in a struct and pass a **pointer** to it as the 6th parameter
- Problem: must validate user pointers. Why? How?
  - Pointer could be NULL (crash OS), or pointing to kernel space (kernel overwriting kernel, security hole!)
- Solution: safe functions to access user pointers:
  - copy_from_user(), copy_to_user(), they make sure the pointer is pointing to user space.

Back to the main plot ...

The problem with **Direct Execution**

➔ Once the program starts to run, the OS becomes a complete outsider -- it doesn't have any **control** over the running program

- The program can access anything it wants to, including doing **restricted operations**.
- The program may never want to **switch** to a different process, which fails the idea of **time sharing**.

We solved Problem #1, now solve #2

Solve Problem #2
Switching Between Processes

The Problem

- If we let a process run completely freely (without control), the process may never stop or switch to another process, then no **time sharing** can be done.
- So OS need to control the process, the question is **how**.
- A simple thought
  - Just let OS start / stop a process as it wants, simple right? What's the problem?
  - Not that simple. While another process is running, OS is NOT running, so **nothing** can really be done by OS.

Main Question:

How can OS **regain control** of the CPU from a process so that it can switch to another process?

Approach #1: Cooperative Processes

- ➔ OS trusts the processes to behave cooperatively
  - ◆ A process that runs for too long would **voluntarily give up** the CPU and hand control to OS.
- ➔ How does a process do this?
  - ◆ periodically make a system call, named `yield()`.
- ➔ Sometimes also give up **involuntarily**.
  - ◆ when the process raises exceptions, will **trap**

This works , in a UTOPIAN world.

Approach #1: Cooperative Processes

Actually, even processes are cooperative, we can still get stuck in some scenarios. Example?

Process enters infinite loop by accident, and there is **`no yeild()`** call in that loop.

Solution:
Reboot computer.

Fun fact: Exactly this happens to early versions of Mac OS.

Approach #2: Non-Cooperative Processes

The OS has to be able to take control in some way,

and it needs help from its best friend -- hardware.

The solutions is: **Timer Interrupt**.

- ➔ A timer device can be programmed to raise an **interrupt** periodically (e.g., every 50 ms)
- ➔ When interrupt is raised, the running process stops, a preconfigured **interrupt handler** in the OS runs.
- ➔ **OS regains control**.

Remember **interrupt** from CSC258?

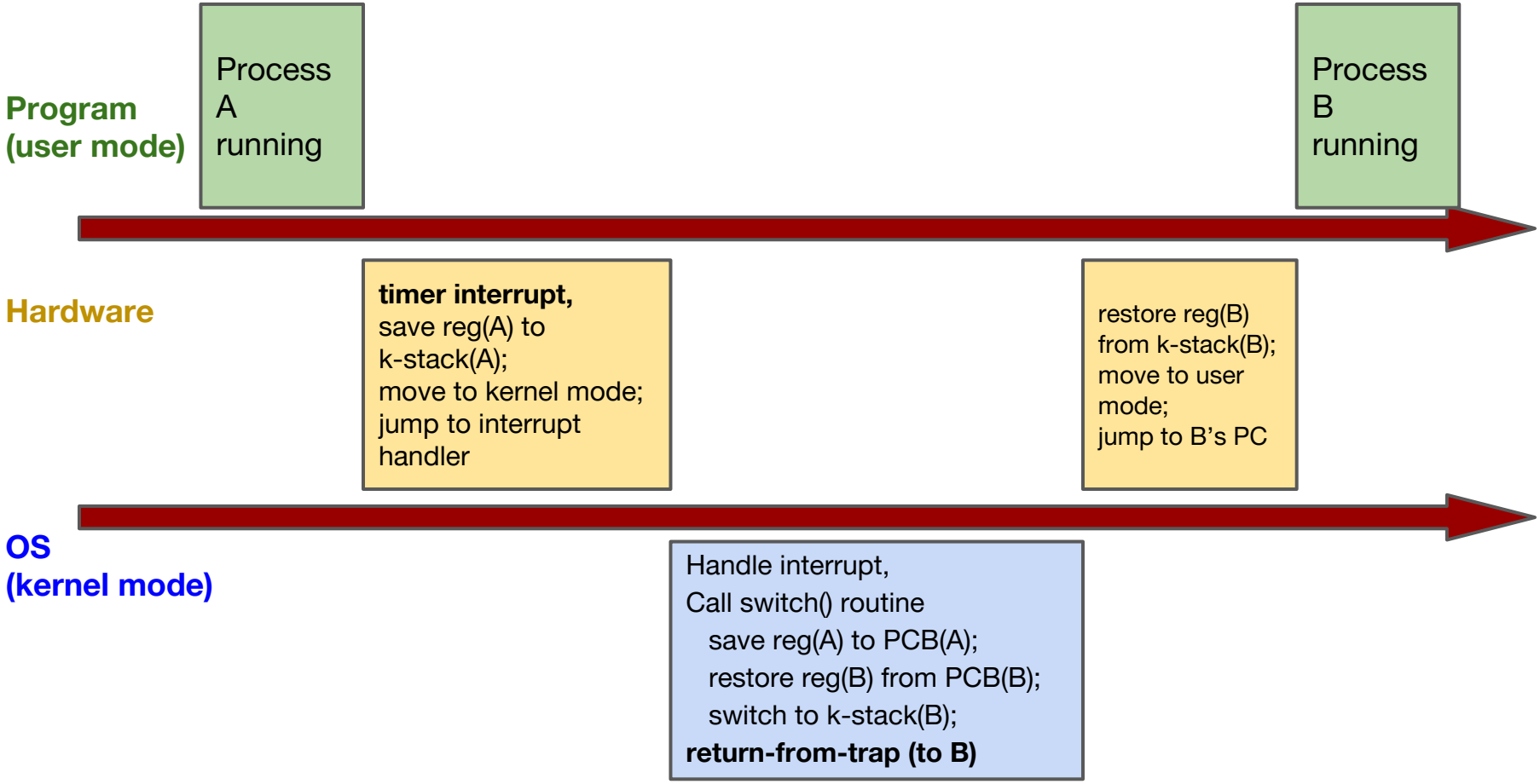
- ➢ It is a **hardware** signal that causes CPU to jump to predefined instructions called **interrupt handler**.
- ➢ Interrupts interact with CPU at very low level, and surpass higher level interactions.

Good, now OS has control.

How to switch to another process then?

- ➔ The OS first decides which process to switch to
  - ◆ done by the **scheduler** (learn about it later)
- ➔ OS executes a piece of assembly code, which we refer to as **context switch**.
  - ◆ **save** register values of **currently-running** process into its kernel stack.
  - ◆ **restore** register values of the **soon-running** process from its kernel stack

Context Switch (from Process A to B)



Problem #2
Switching Between Processes

Summary

The challenge: **efficiently** virtualize CPU with **control.**

Mechanism: **time sharing** with **limited direct execution.**

➔ Process can perform restrict operations without messing around with hardware

- System calls

➔ OS can switch from one process to another

- Cooperative vs noncooperative
- Timer interrupt
- Context switch

Based on what we have learned so far, what are some of the things that the a computer / OS need to do while **booting**?

- ➔ Initialize hardware
- ➔ Initialize system call table, interrupt descriptor table
- ➔ Create the first process (init)
- ➔ Starting in kernel mode, switch to user mode and start first user process.
- ➔ Timer device start generating timer interrupts

Next week

➔ Thread

➔ Concurrency

This week’s tutorial

➔ Intro to kernel programming