OS organization part I:
the first 3 steps of building an operating system

Step1-3 of building an OS

- Step #3: understand computer architecture

- Step #2: understand interrupt and exception

- Step #1: understand context-switch

Step1-3 of building an OS

➡ Step #3: understand computer architecture

- memory layout

- running a program

- calling convention

- Step #2: understand interrupt and exception

- Step #1: understand context-switch

Before building an OS, you have

Computer

Hardware documents

Two important documents

From the CPU vendor

From the computer vendor

Your labs, egos-2k+

- RISC-V and SiFive documents
- (which are simpler and shorter than Intel/Dell)

Chapter4 of siFive is memory map

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	
0x0000_1000	0x0000_1FFF	R XC	Mode Select	
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4: FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

CPU debug @0x0000_0000
(ignore this for your labs)

Device control @0x0200_0000

Boot ROM @0x2000_0000

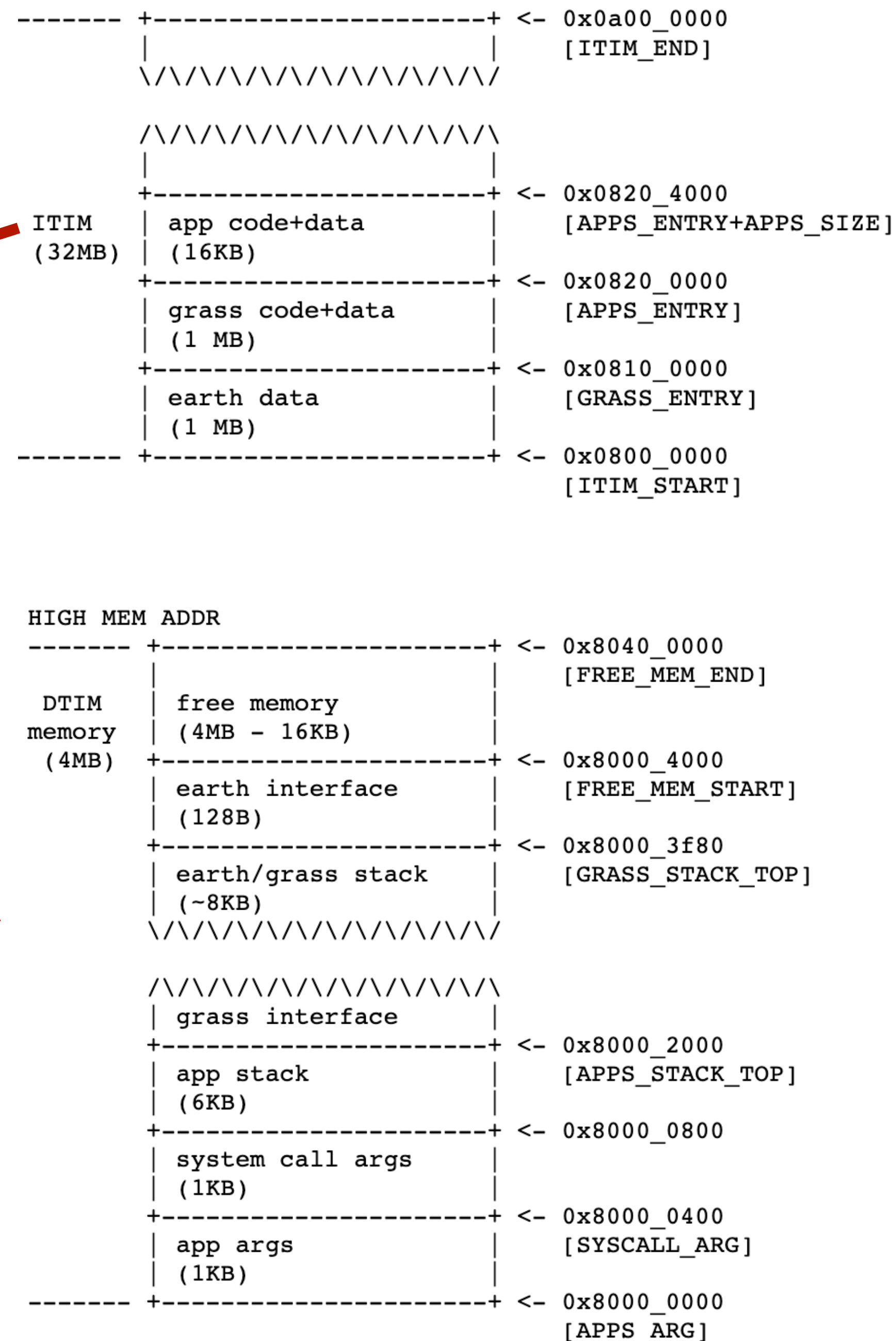Main memory @0x8000_0000

(main memory ≤2GB in this architecture)

How does egos-2k+ use the memory?

(Read "egos-2k+ memory layout" on the Reference page)

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	On-Chip Non Volatile Memory
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	On-Chip Peripherals
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4: FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics


```
------- +--------------------+ <- 0x0a00_0000
        |                    |    [ITIM_END]
        \/\/\/\/\/\/\/\/\/\/\/

        /\/\/\/\/\/\/\/\/\/\/\
        |                    |
        +--------------------+ <- 0x0820_4000
ITIM    | app code+data      |    [APPS_ENTRY+APPS_SIZE]
(32MB)  | (16KB)             |
        +--------------------+ <- 0x0820_0000
        | grass code+data    |    [APPS_ENTRY]
        | (1 MB)             |
        +--------------------+ <- 0x0810_0000
        | earth data         |    [GRASS_ENTRY]
        | (1 MB)             |
------- +--------------------+ <- 0x0800_0000
                                  [ITIM_START]


HIGH MEM ADDR
------- +--------------------+ <- 0x8040_0000
        |                    |    [FREE_MEM_END]
DTIM    | free memory        |
memory  | (4MB - 16KB)       |
(4MB)   +--------------------+ <- 0x8000_4000
        | earth interface    |    [FREE_MEM_START]
        | (128B)             |
        +--------------------+ <- 0x8000_3f80
        | earth/grass stack  |    [GRASS_STACK_TOP]
        | (~8KB)             |
        \/\/\/\/\/\/\/\/\/\/\/

        /\/\/\/\/\/\/\/\/\/\/\
        | grass interface    |
        +--------------------+ <- 0x8000_2000
        | app stack          |    [APPS_STACK_TOP]
        | (6KB)              |
        +--------------------+ <- 0x8000_0800
        | system call args   |
        | (1KB)              |
        +--------------------+ <- 0x8000_0400
        | app args           |    [SYSCALL_ARG]
        | (1KB)              |
------- +--------------------+ <- 0x8000_0000
                                  [APPS_ARG]
```
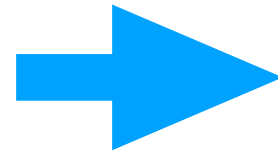
Step1-3 of building an OS

➡ Step #3: understand computer architecture

  ✅memory layout

  • running a program

  • calling convention

• Step #2: understand interrupt and exception
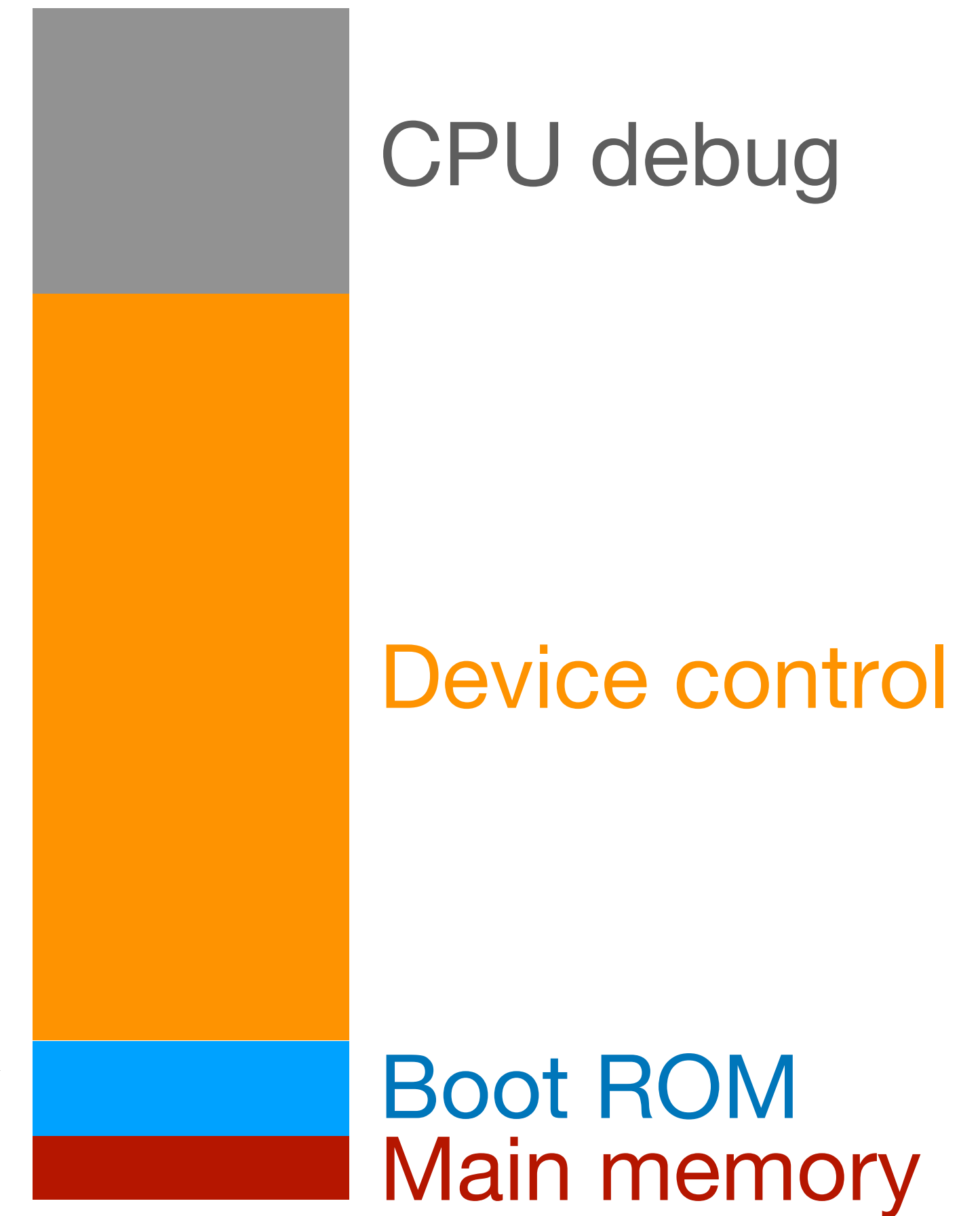
• Step #1: understand context-switch

provides a **hello world**

Step#1: compile the hello-world program in Linux

Step#2: copy the compiled code to the boot ROM

with special tools provided by SiFive

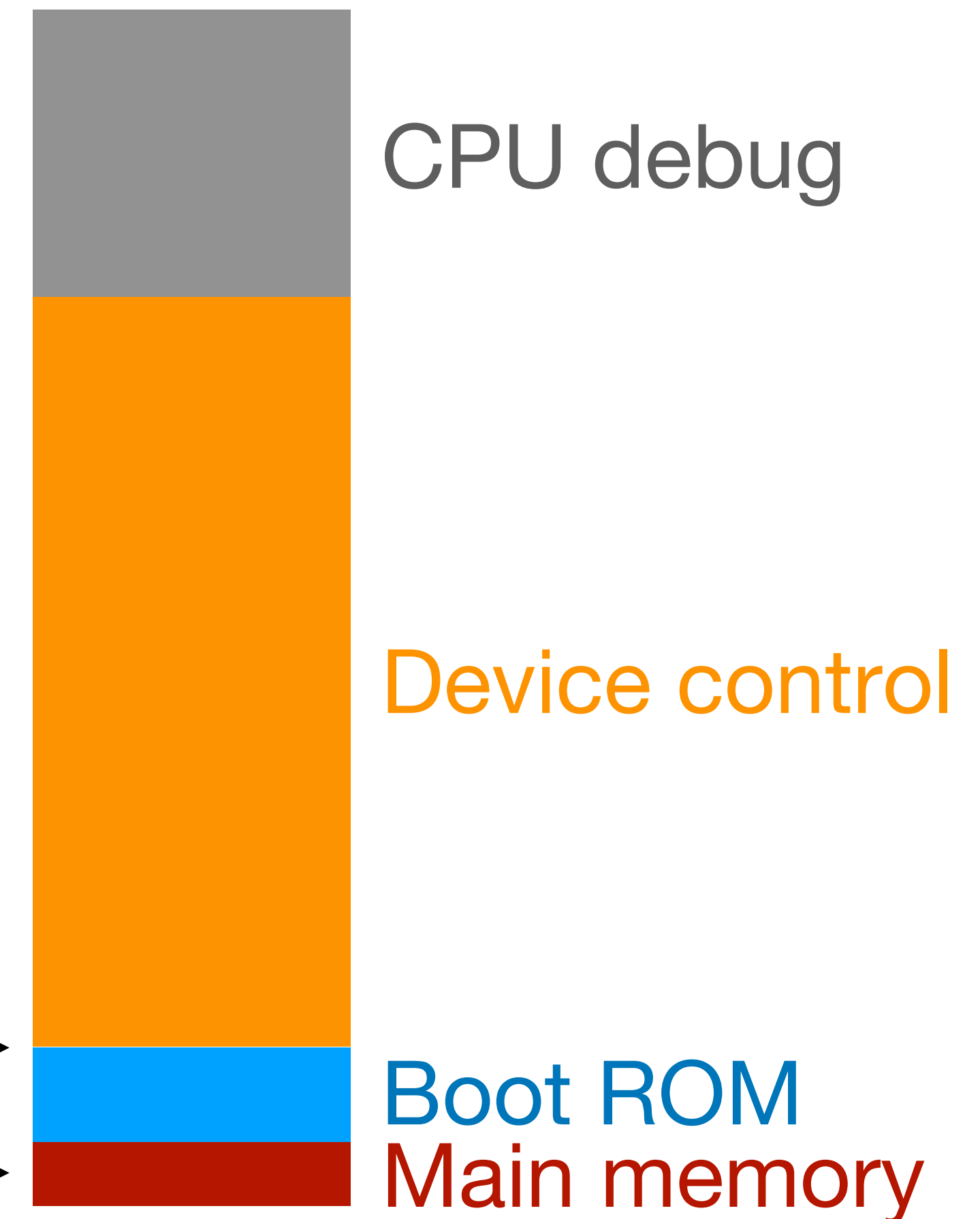CPU debug

Device control

Boot ROM
Main memory

Enter the context of hello world

Step#3: press the boot button on the computer

Step#4: CPU set instruction pointer to the beginning of boot ROM

Step#5: an `li` instruction sets the stack pointer to main memory

CPU debug

Device control

Boot ROM
Main memory

hello-world prints to screen

"Hello World"

Step#8: the screen shows "Hello World!"

CPU debug

Step#7: during `printf()`, `store` instructions will send data to the screen

Device control

Step#6: a `call` instruction calls `main()` which calls `printf()`

Boot ROM
Main memory
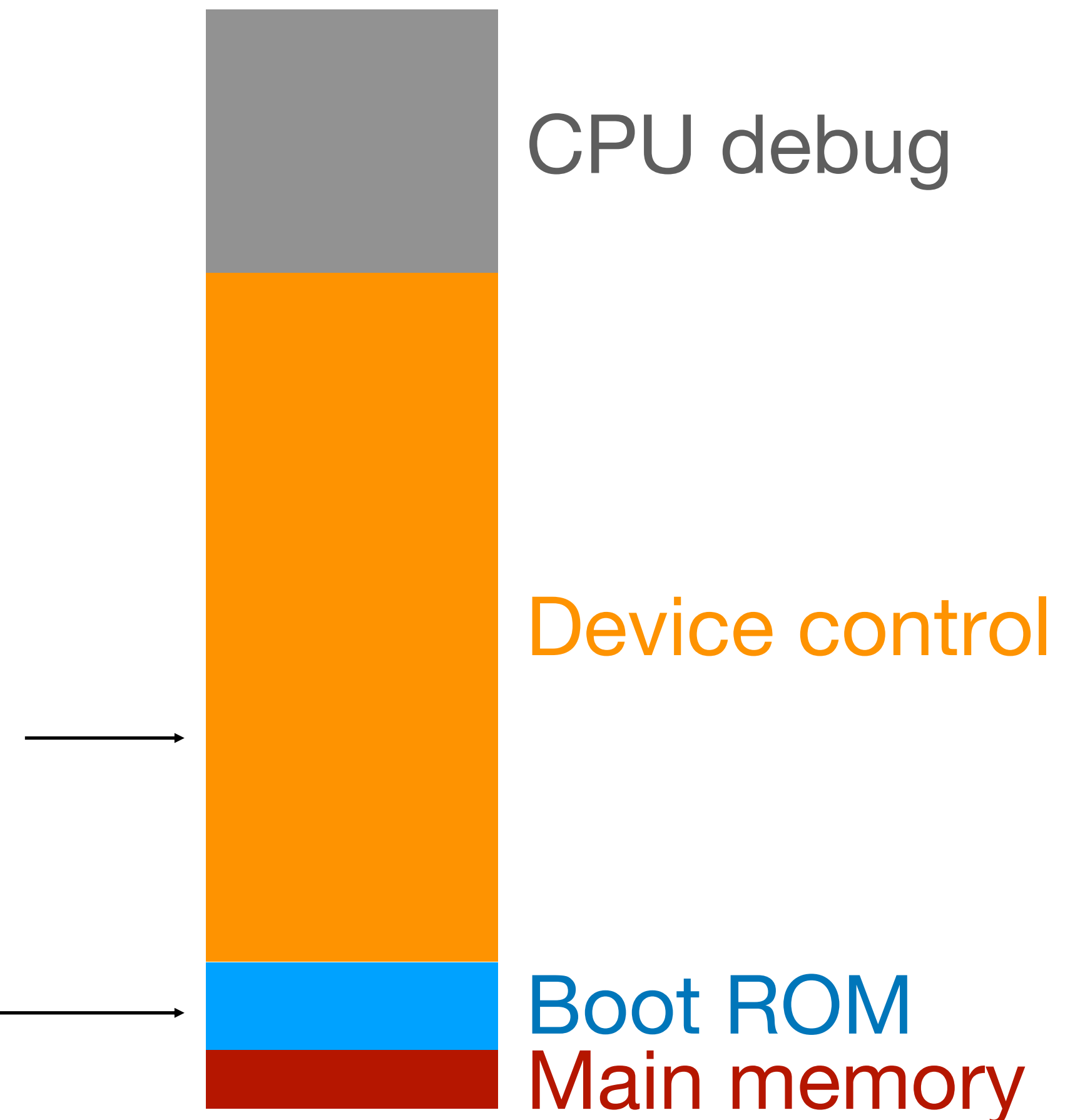
Question: Where (in memory) is the first kernel instruction executed by CPU?

(use gdb to tell)

Question:
How does printf() work?

Step1-3 of building an OS

➡️ Step #3: understand computer architecture

   ✅ memory layout

   ✅ running a program

   • calling convention

• Step #2: understand interrupt and exception

• Step #1: understand context-switch

Calling convention in RISC-V

Register	ABI Name	Description	Saver	
x0	zero	Hard-wired zero	—	
x1	ra	Return address	Caller	main()
x2	sp	Stack pointer	Callee	printf()
x3	gp	Global pointer	—	
x4	tp	Thread pointer	—	
x5	t0	Temporary/alternate link register	Caller	main()
x6–7	t1–2	Temporaries	Caller	main()
x8	s0/fp	Saved register/frame pointer	Callee	printf()
x9	s1	Saved register	Callee	printf()
x10–11	a0–1	Function arguments/return values	Caller	main()
x12–17	a2–7	Function arguments	Caller	main()
x18–27	s2–11	Saved registers	Callee	printf()
x28–31	t3–6	Temporaries	Caller	main()

Table 25.1 of RISC-V manual, volume1

Function call step#1
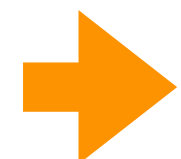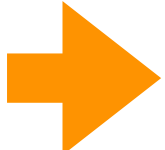
```
<main>:

    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡)
  ➡ Restore caller-saved registers

    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#2

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡ )
 ➡  Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```
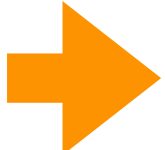
Function call step#3
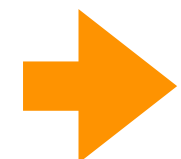
```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡ )
  ➡ Restore caller-saved registers
    . . .


<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#4

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡ )
➡   Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#5

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡ )
➡   Restore caller-saved registers
    . . .


<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Function call step#6

```
<main>:
    . . .
    Store caller-saved registers on the stack
    Call printf (set ra to the address of ➡ )
 ➡  Restore caller-saved registers
    . . .

<printf>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to main() (set pc to ra)
```

Step1-3 of building an OS

➡️ Step #3: understand computer architecture

✅ memory layout

✅ running a program

✅ calling convention

- Step #2: understand interrupt and exception

- Step #1: understand context-switch

Question: Can we simultaneously run multiple helloworlds?

Yes, time-sharing/multiplexing CPUs;
namely, adding a timer handler.

Step1-3 of building an OS

- Step #3: understand computer architecture

➡ Step #2: understand interrupt and exception

  - control and status registers (CSR)

  - "inserting" a call to the handler function

- Step #1: understand context-switch and multi-threading

Control and status registers (CSR)

- There are **many** registers other than the 32 user-level ones:

  - `misa`: 32-bit or 64-bit?

  - `mhartid`: the core ID number

  - `mstatus`: the machine status

  - `mtvec`, `mie`, `mtime`, `mtimecmp`: interrupt handling

Recap: timer interrupt

- How to register an interrupt handler?

  - write the address of `handler` function to `mtvec`

- How to set a timer?

  - write (`mtime` + `QUANTUM`) to `mtimecmp`

- How to enable timer interrupt?

  - set certain bit of `mstatus` and `mie` to 1

Recap: a timer handler program

```c
int quantum = 50000;

void handler() {
    earth->tty_info("Got timer interrupt.");
    mtimecmp_set(mtime_get() + quantum);
}

int main() {
    earth->tty_success("A timer interrupt example.");

    asm("csrw mtvec, %0" ::"r"(handler));
    mtimecmp_set(mtime_get() + quantum);

    int mstatus, mie;
    asm("csrr %0, mstatus" : "=r"(mstatus));
    asm("csrw mstatus, %0" ::"r"(mstatus | 0x8));
    asm("csrr %0, mie" : "=r"(mie));
    asm("csrw mie, %0" ::"r"(mie | 0x80));

    while(1);
}
```

Set a timer

Register handler
Set a timer

Enable timer interrupt

What do interrupts look like from a CPU's point of view?

Assume an interrupt

```
<some user function>:
    . . .
    <interrupt will happen here>
    . . .



<handler>:
    . . .
```

Intuition: CPU/OS "inserts" these code

```
<some user function>:
    . . .
    Store caller-saved registers on the stack
    Call handler (set ra to the address of ➡ )
 ➡  Restore caller-saved registers
    . . .


<handler>:
    Store callee-saved registers on the stack
    . . .
    Restore callee-saved registers
    Return to some_user_function() with ra
```

Cleanup these code

```
<some user function>:
    . . .
    Store caller-saved registers on the stack
    Call handler (set ra to the address of ➡ )
    Restore caller-saved registers
 ➡  . . .

<handler>:
    Store all registers on the stack
    . . .
    Restore all registers
    Return to some_user_function() with ra
```

Handler returns to the same context

```
<some user function>:
    . . .
    Call handler (set ra to the address of ➡ )
➡  . . .


<handler>:
    Store all registers on the stack
    . . .
    Restore all registers
    Return to some_user_function() with ra
```
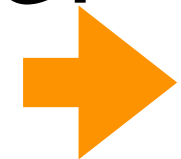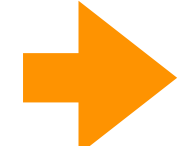
Question

How does the handler function switch to the context to a different process?

First, replacing *ra* with CSR mepc

```
<some user function>:
    . . .
    // mepc: machine exception program counter
    Call handler (set mepc to the address of ➡ )
➡   . . .


<handler>:
    Store all registers on the stack
    . . .
    Restore all registers
    Return to some_user_function() with mepc
```
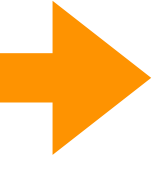
Then, switch context with mepc

```
<some user function>:
    . . .
    Call handler (set mepc to the address of ➡ )
➡  . . .

<handler>:
    Store all registers on the stack
    . . .
    Set mepc to the code section of another thread
    Restore all registers
    Switch to another process with mepc
```

A demo using mepc and mret

```c
void thread0() { while(1) { // print something green } }
void thread1() { while(1) { // print something yellow } }

int next_thread = 0;
void handler() {
    next_thread = 1 - next_thread;
    asm("csrw mepc, %0" ::"r"((next_thread == 0)? thread0 : thread1));

    mtimecmp_set(mtime_get() + quantum);        // reset timer
    asm("li sp, 0x80002000");                   // set stack pointer
    asm("mret");        // forget previous thread and start a new thread
}
```

Brief summary

- The interrupt handler function

  - Stores `all` register on stack, instead of callee-saved

  - Uses `mret` and `mepc` instead of `ret` and `ra`

- This is why, in the demo code, there is one line:

  - void handler() __attribute__((interrupt));

  - telling the compiler this function is an interrupt hander

Question: There are three ways to trap to kernel. What are they?

They are interrupts, exceptions, and syscalls.

Kernel ≈ 3 handlers (*)

```c
void kernel() {  // registered to CSR mtvec
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;    // take the last 10 bits
    if (mcause & (1 << 31)) {   // most significant bit is 1?
        if (id == 7) { timer_handler(); }
    } else {
        if (id == 8) { syscall_handler(); }
                else { fault_handler(); }
    }
}
```

CPU view of a running computer
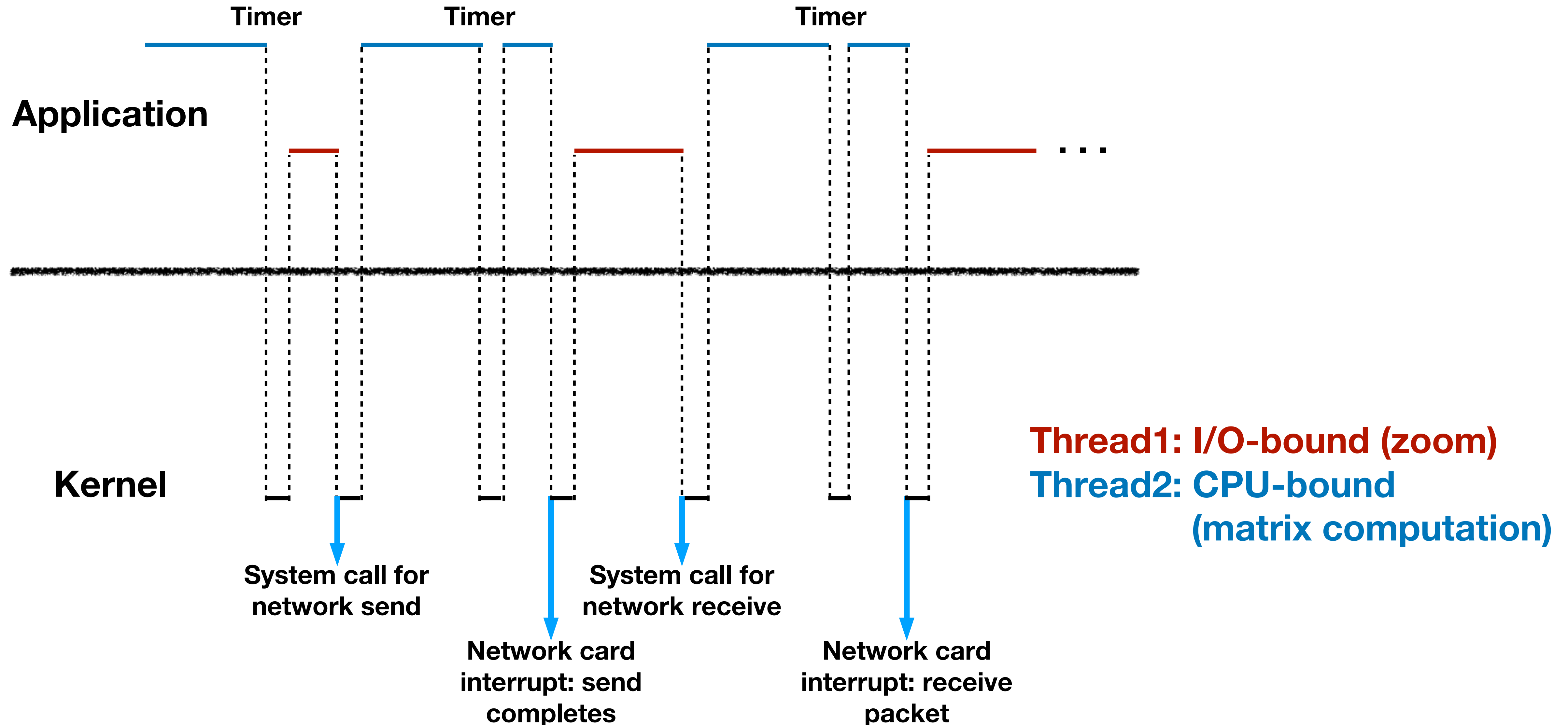
Step1-3 of building an OS

- Step #3: understand computer architecture

➡️ Step #2: understand interrupt and exception

 ✅ control and status registers (CSR)

 ✅ "inserting" a function call to the handler

- Step #1: understand context-switch and multi-threading

Step1-3 of building an OS

➡️ Step #3: understand computer architecture

- memory layout

- running a program

- calling convention

- Step #2: understand interrupt and exception

- Step #1: understand context-switch

Step1-3 of building an OS

- Step #3: understand computer architecture

➡ Step #2: understand interrupt and exception

  - control and status registers (CSR)

  - "inserting" a call to the handler function

- Step #1: understand context-switch and multi-threading

Step1-3 of building an OS

- Step #3: understand computer architecture

- Step #2: understand interrupt and exception

➡ Step #1: understand context-switch

  - program context

  - switching from one process to another

# Simplified but not by much:
context = memory abstraction
+ CPU registers

data structures and algorithms

Object
Reference
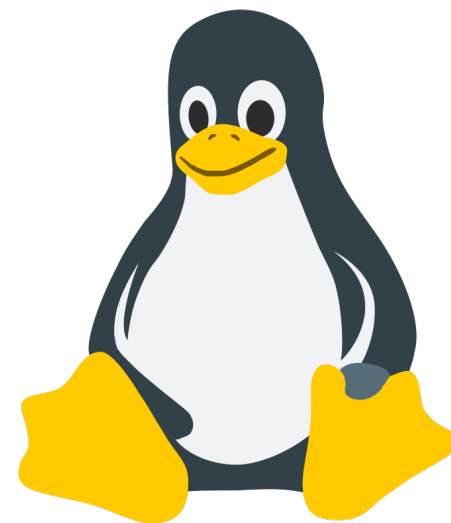
Memory
Pointer
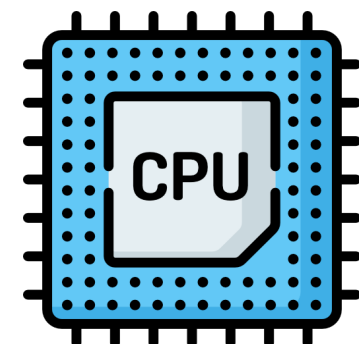
Recall RISC-V asm instructions
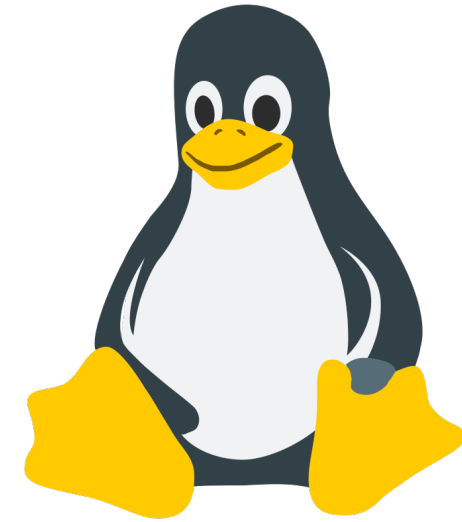
Interface
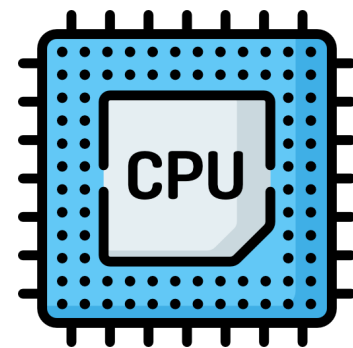
- load / store instructions

- instruction / stack pointer registers

From physics to abstraction

- An ECE course would study voltage, current, etc.

- A CS course studies the abstraction of memory.

  - i.e., a simple math model, such as

Content	1st byte	2nd byte	3rd byte	……	2^32th byte
Address	0x0000 0000	0x0000 0001	0x0000 0002	……	0xFFFF FFFF

A program's view of the memory

```c
int str_len = 14;

int main() {
    char* str = malloc(str_len);
    memcpy(str, "Hello World!\n", str_len);
    printf("%s", str);
    return 0;
}
```
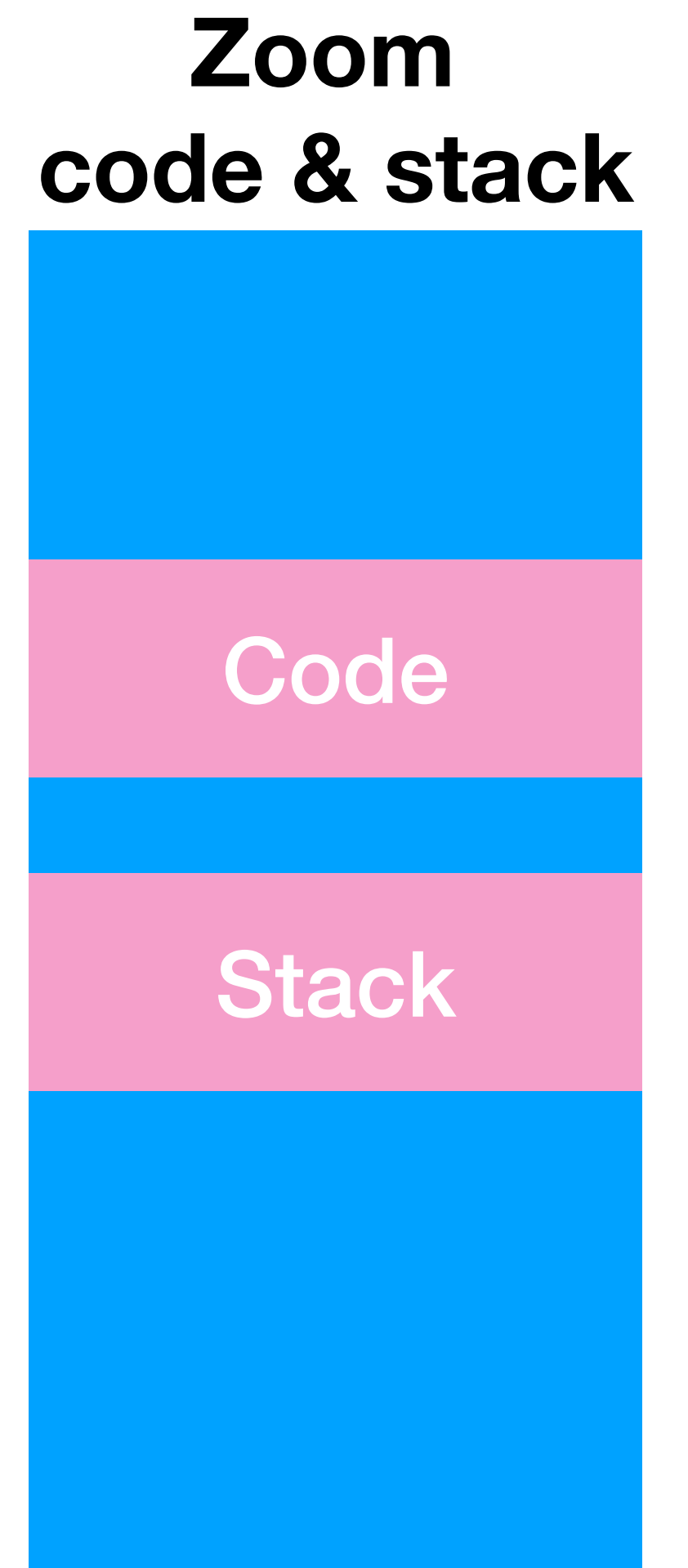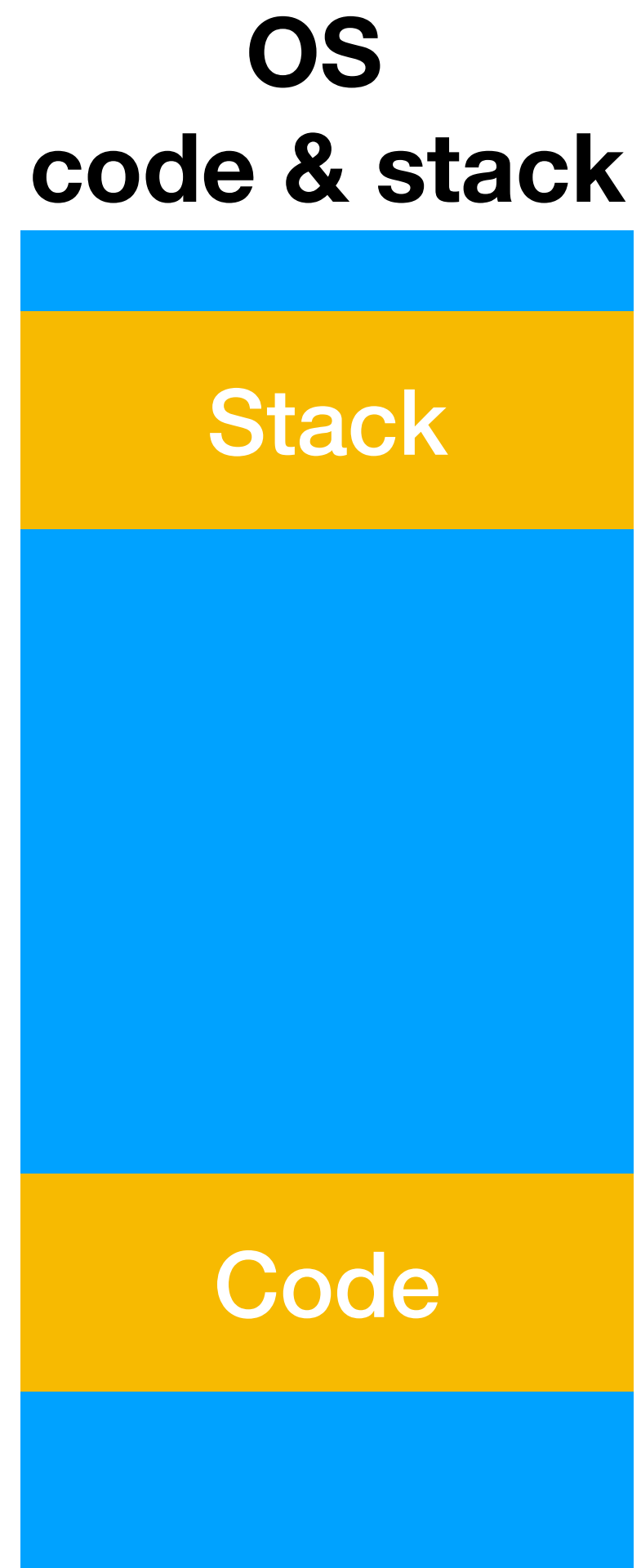
Stack

Heap

Data

Read-only data
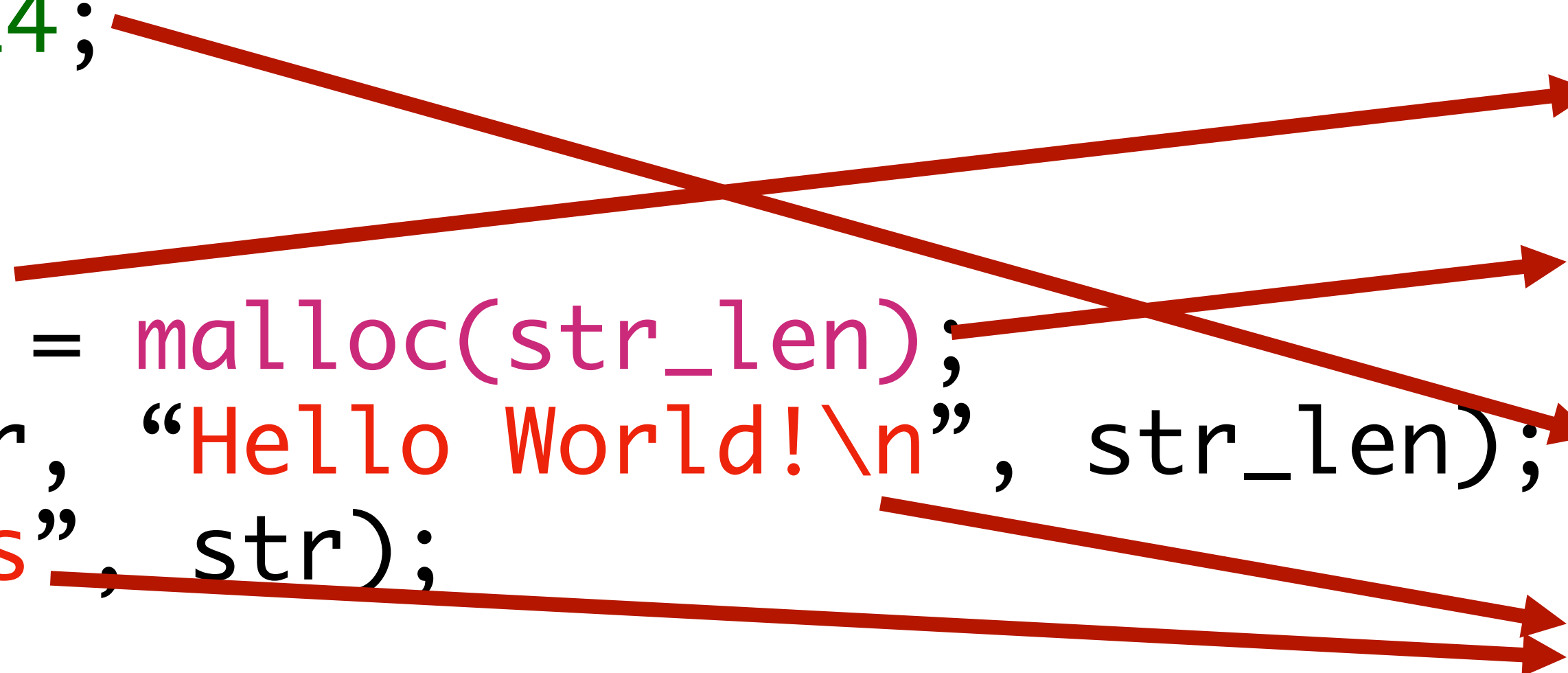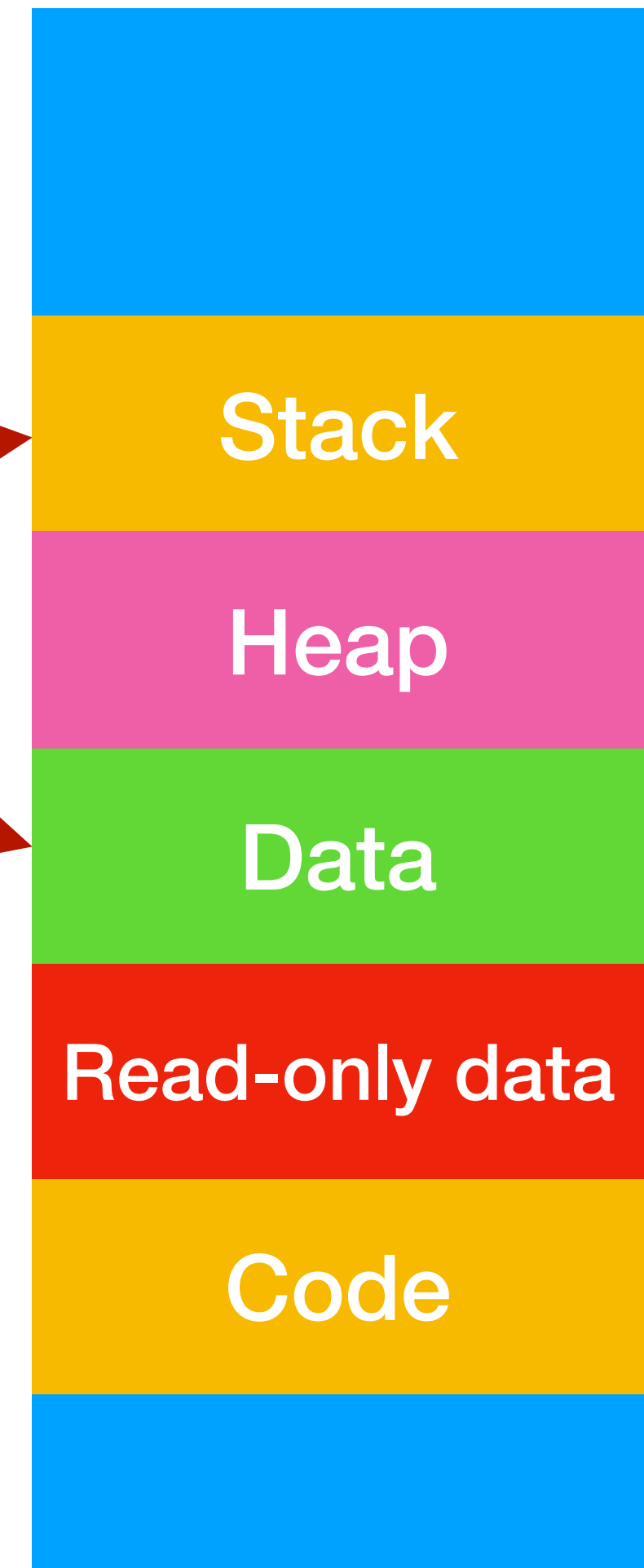
Code

Simplified but not by much:
context = memory abstraction
+ CPU registers

OS is a program

OS
code & stack
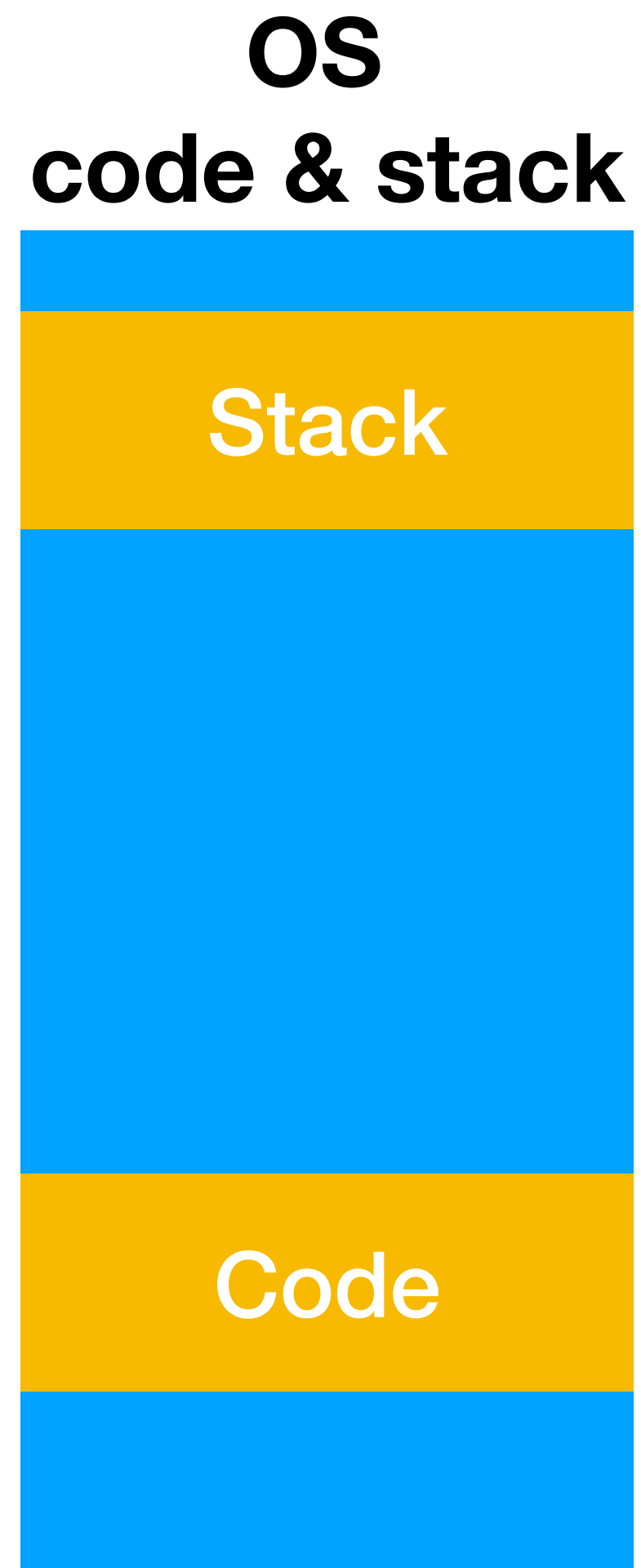
Stack

Code

Zoom is another program

OS
code & stack

Zoom
code & stack

Stack

Code

Code

Stack

Add CPU into the picture

OS
code & stack

Stack

Code

CPU

Stack pointer register

Instruction pointer register

Zoom
code & stack

Code

Stack

CPU in the **context** of **OS**

OS
code & stack

Stack

Code

CPU

Stack pointer register

Instruction pointer register

Zoom
code & stack

Code

Stack

CPU in the context of Zoom

OS code & stack

- Stack
- Code

CPU

- Stack pointer register
- Instruction pointer register
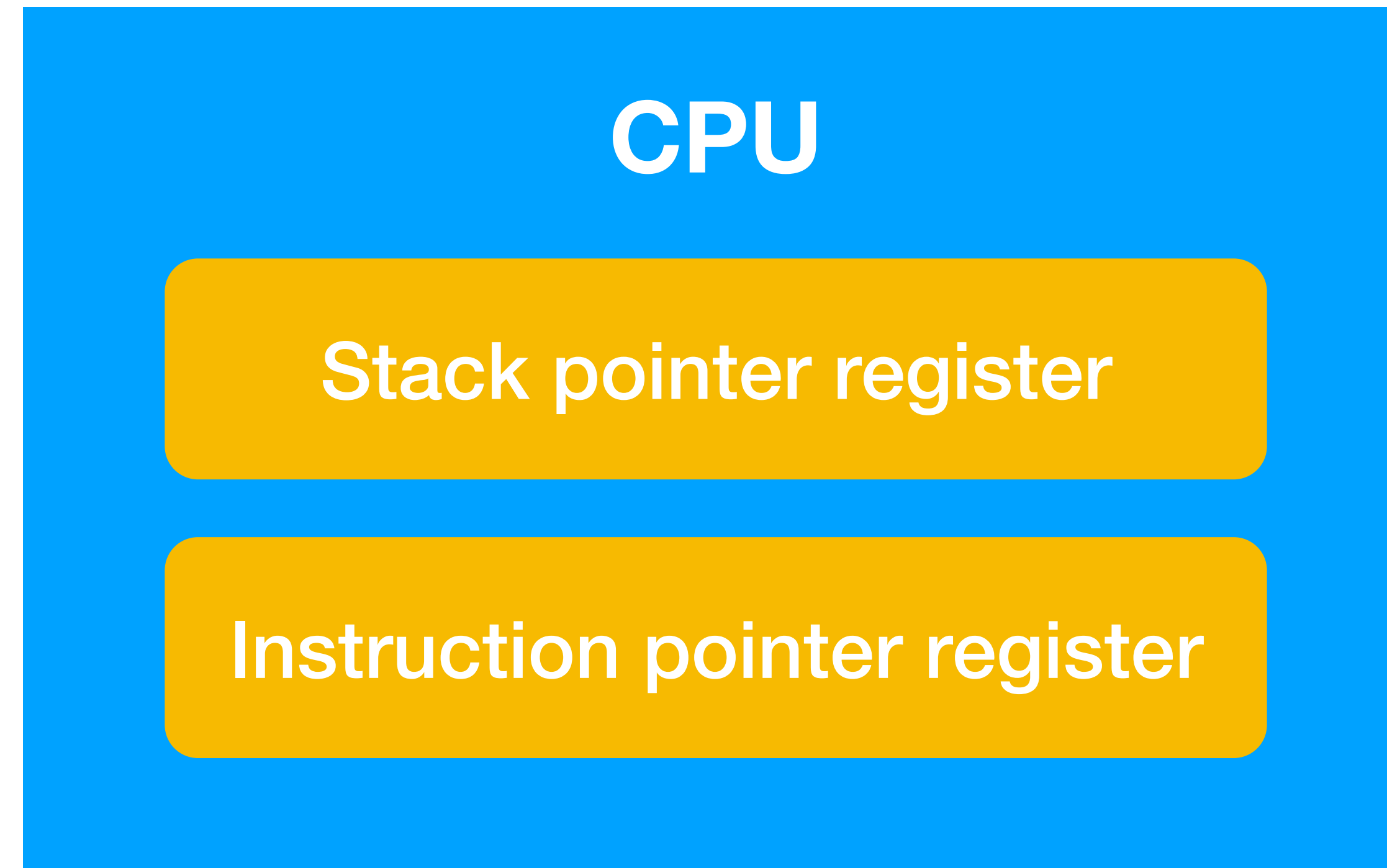
Zoom code & stack

- Code
- Stack
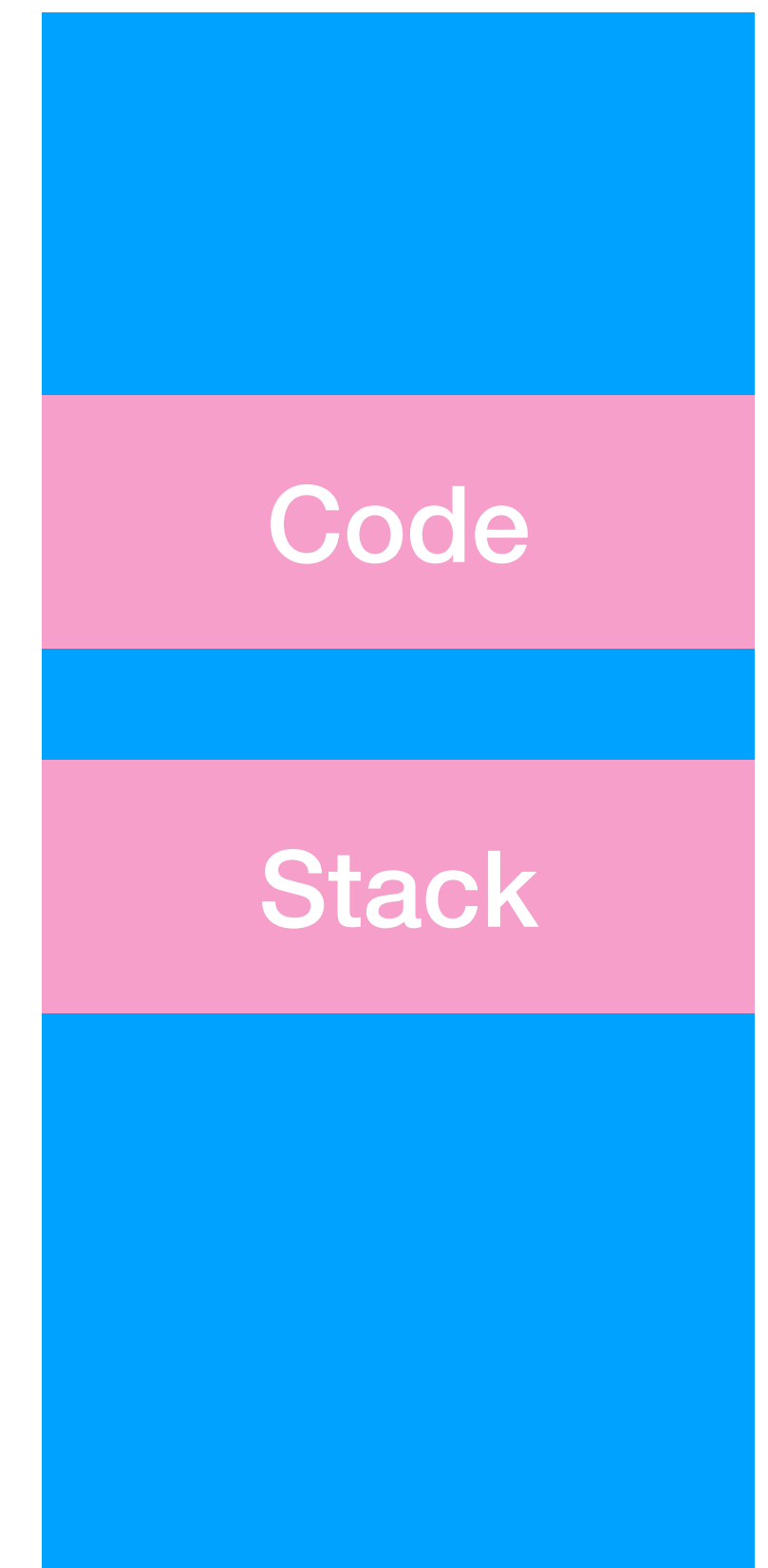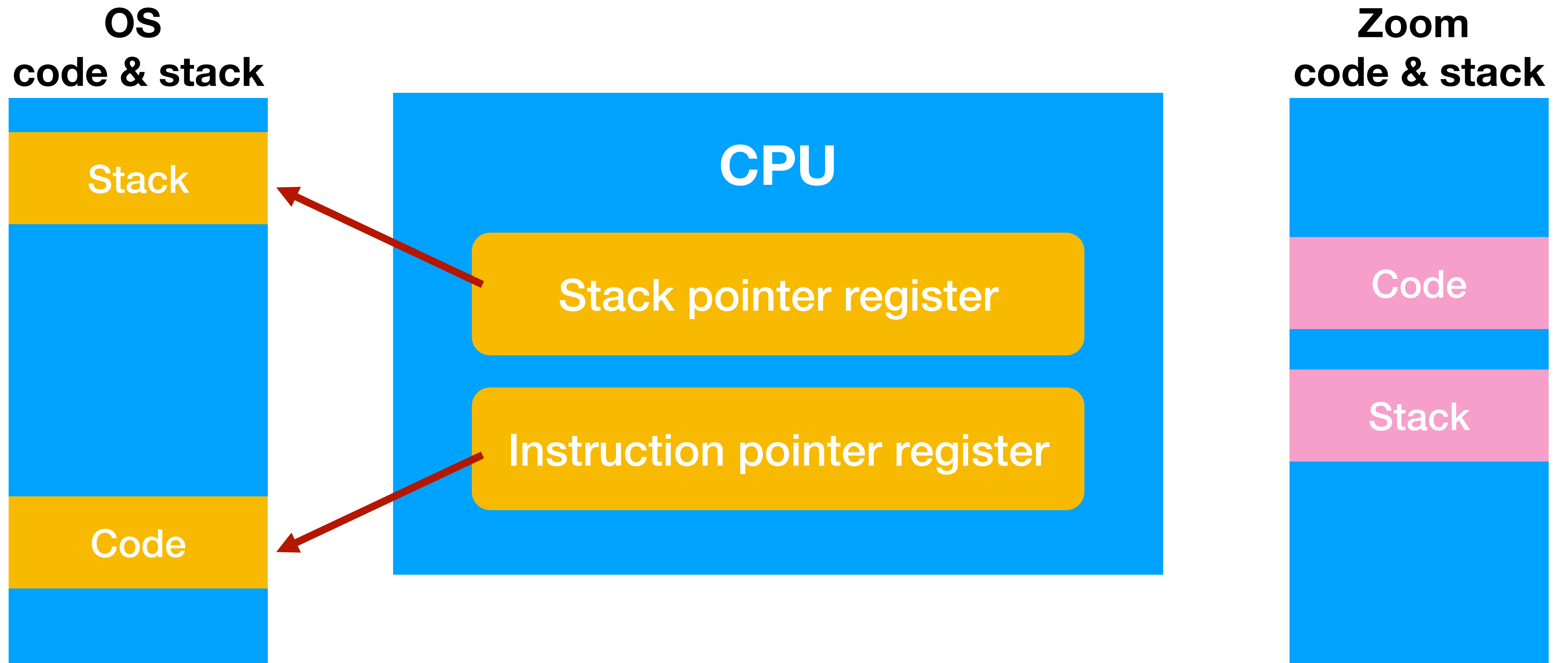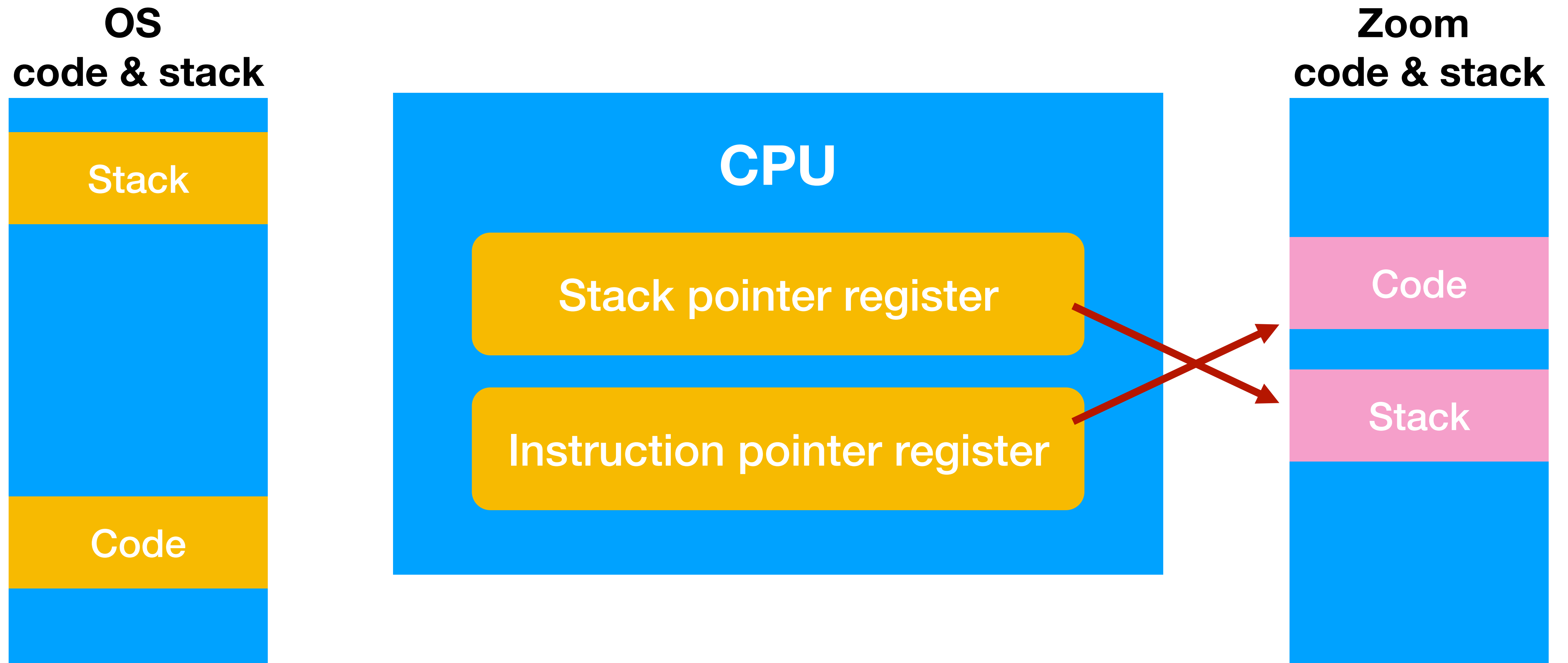
Memory view in practice (before virtual memory)

physical memory

CPU

Stack pointer register

Instruction pointer register

Stack

Code

Stack

Code
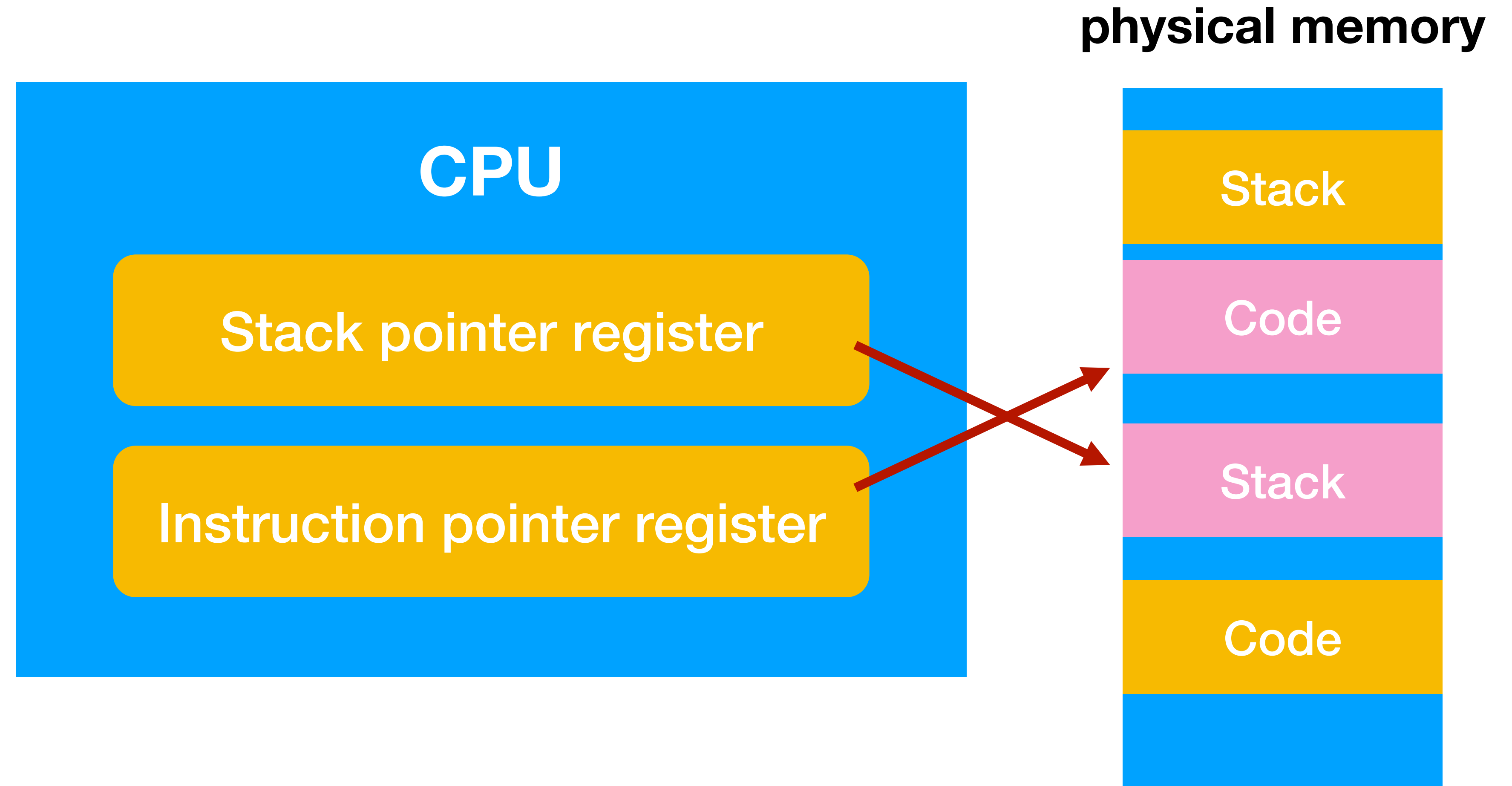
Memory view in practice (after virtual memory)
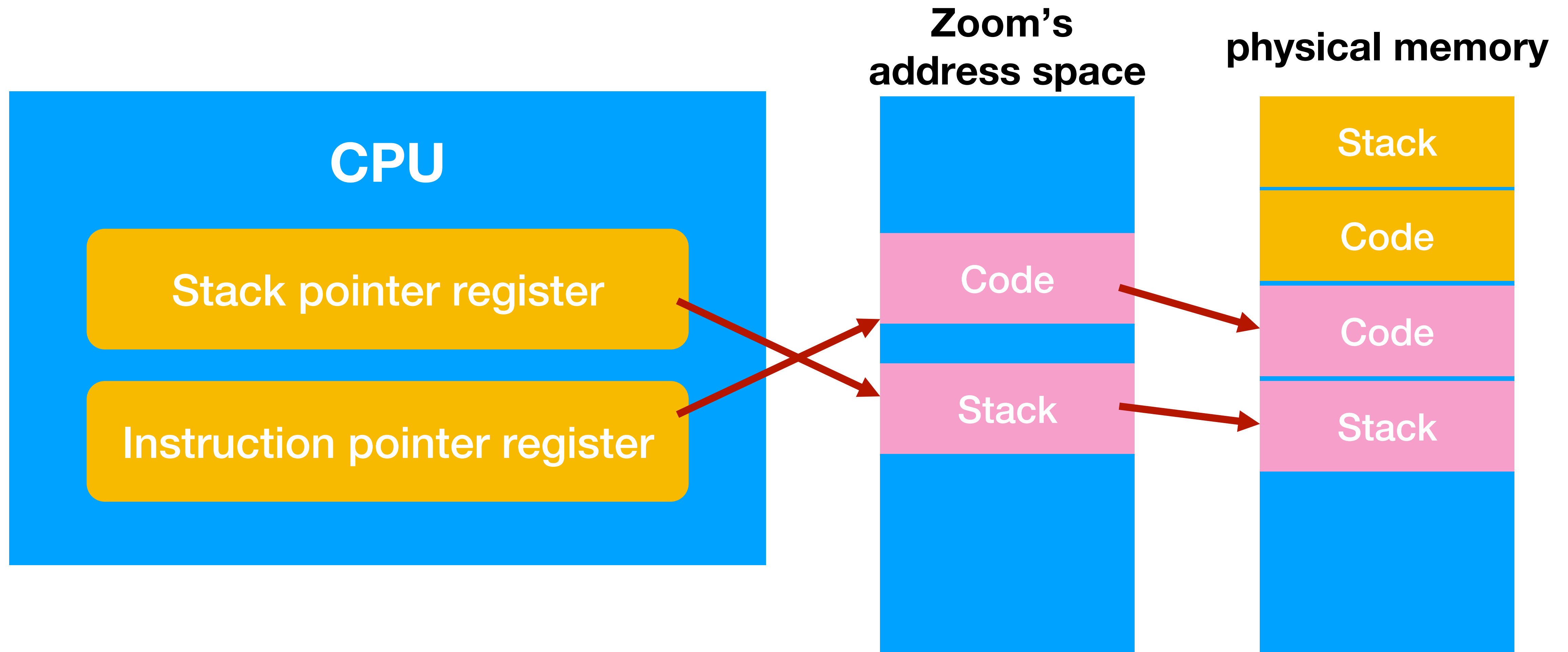
Zoom's address space

physical memory

CPU

Stack pointer register

Instruction pointer register

Code

Stack

Stack

Code

Code

Stack

Step1-3 of building an OS

- Step #3: understand computer architecture

- Step #2: understand interrupt and exception

➡️ Step #1: understand context-switch

✅program context

- switching from one process to another

Switching from one process to another

- Context switch in egos-2k+:

  - switch the memory abstraction from one to another
    (read earth/cpu_mmu.c)

  - switch the CPU registers from one to another
    (read grass/kernel.c)

Next, about OS organization