Memory Management: Translation and Protection

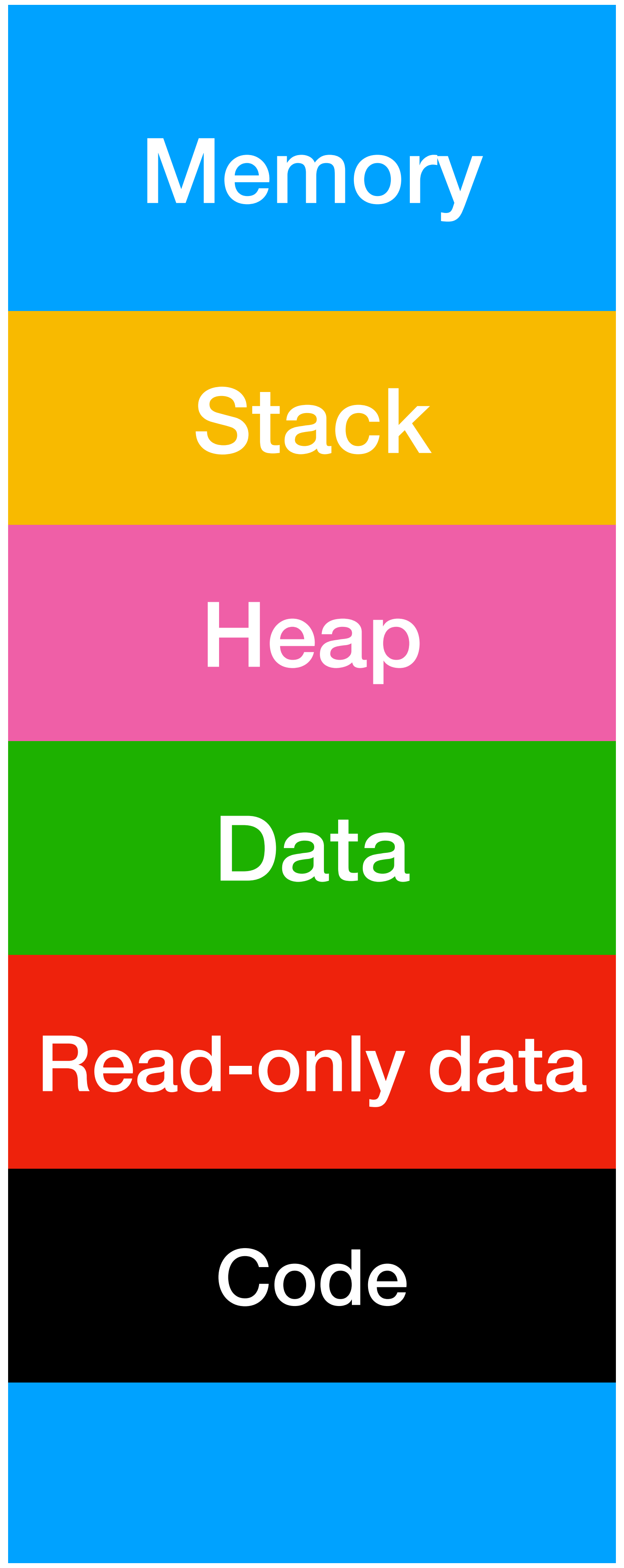
Memory Management

- ➔ Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  - Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory

Recall hello-world from week#2

```
int str_len = 14;

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



Recall memory map from week#6

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 @0000_0000
(ignore this for building an OS)

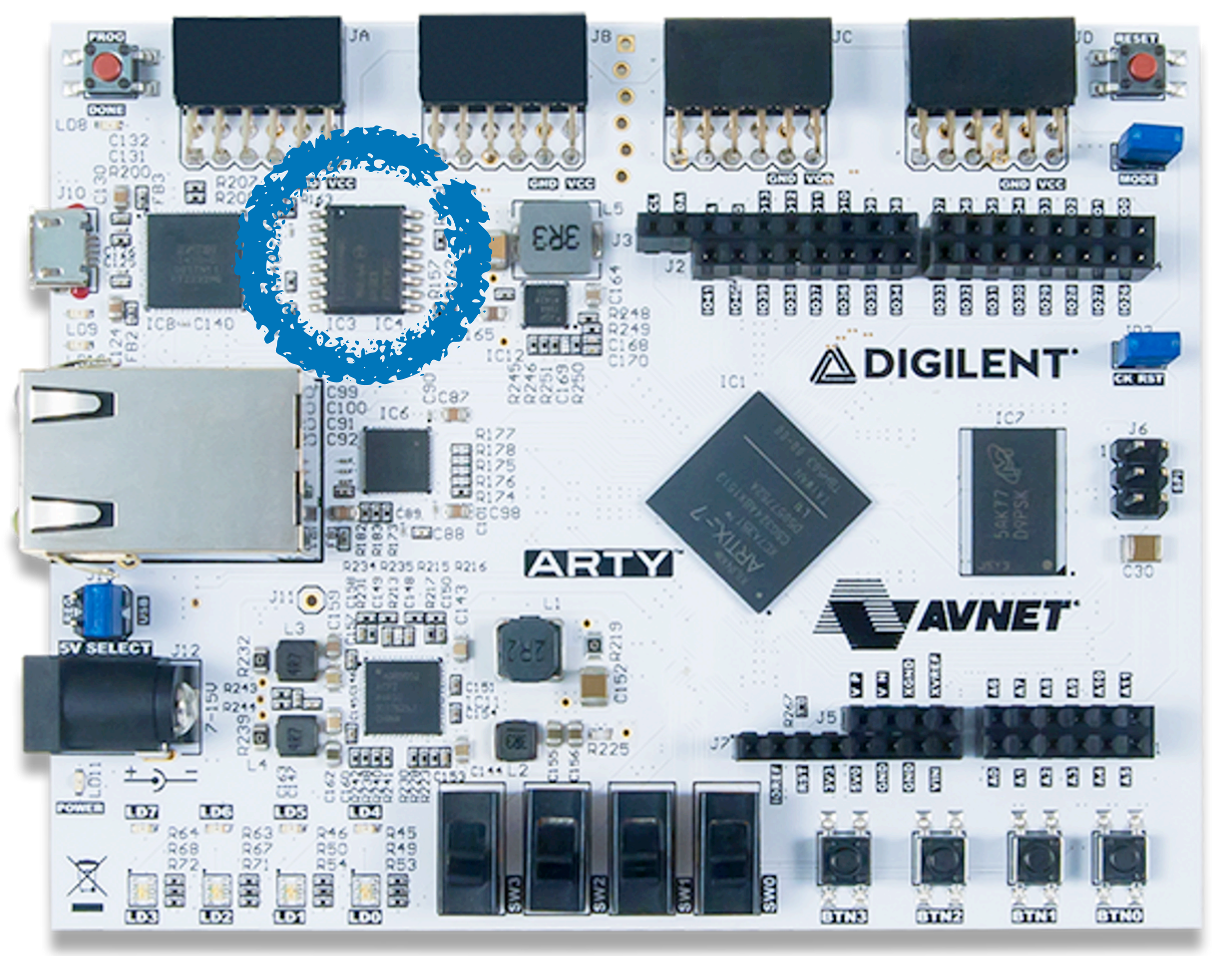
Device control @0200_0000

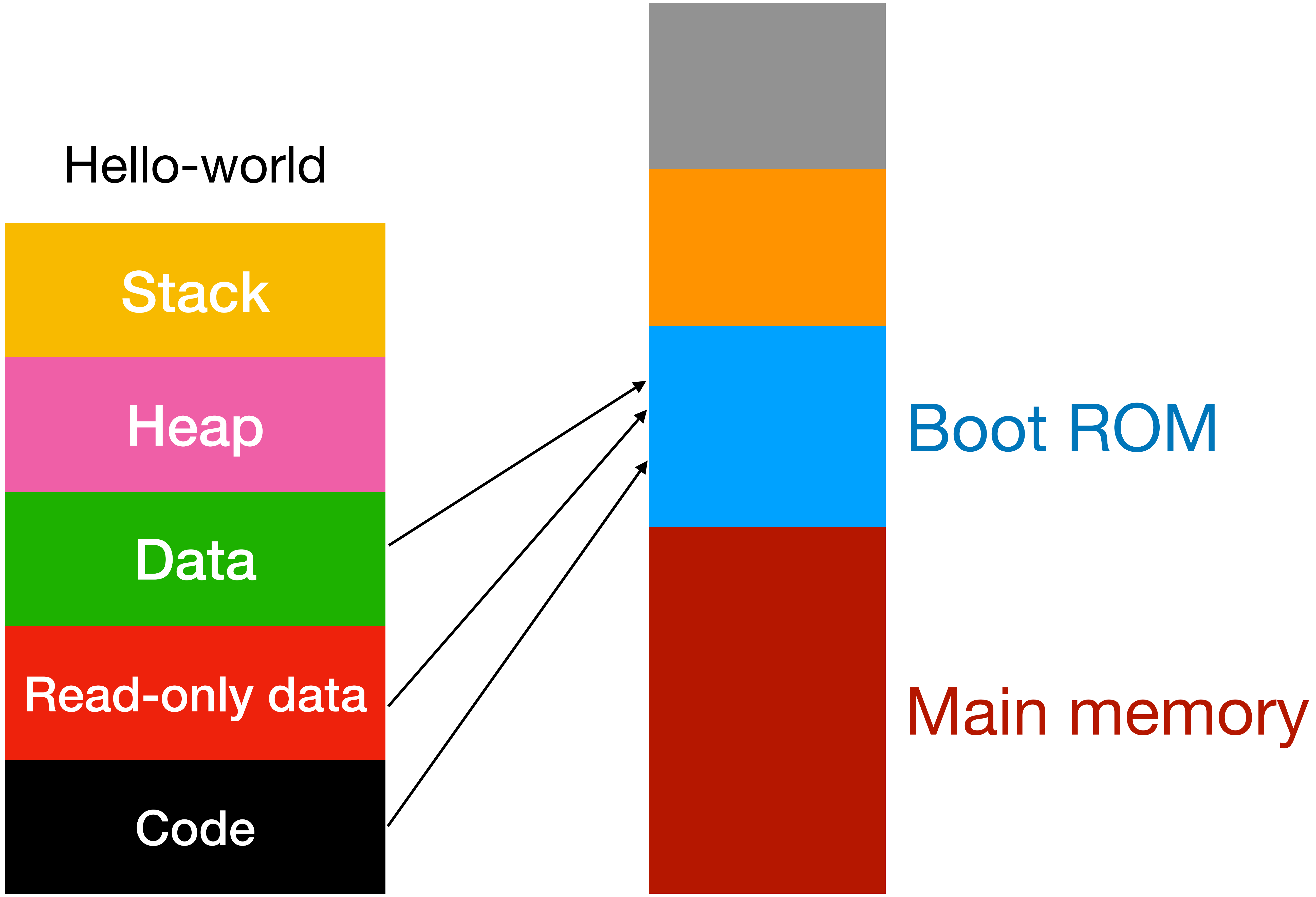
Boot ROM @2000_0000

Main memory @0x8000_0000

Before boot up

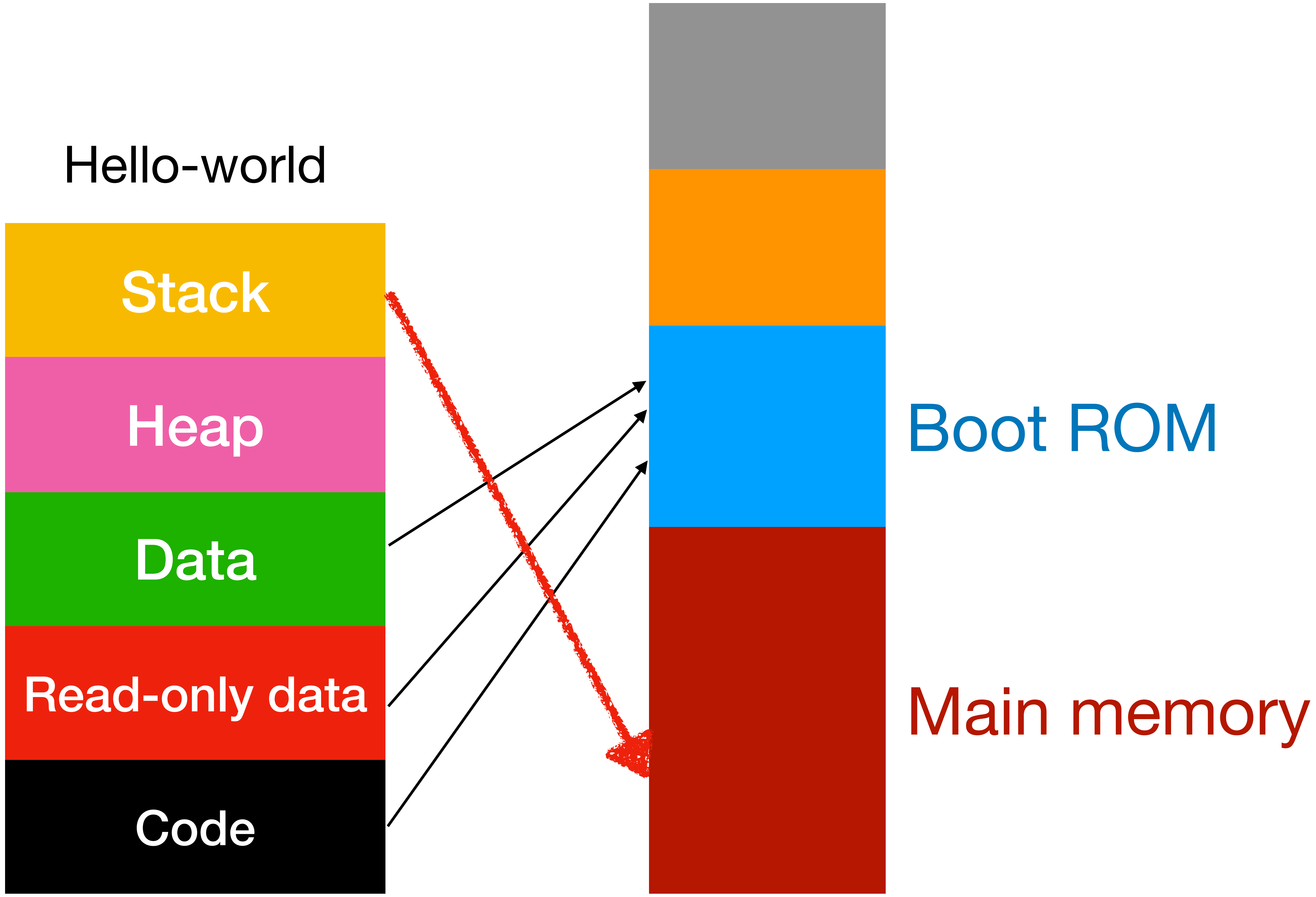
- Boot ROM holds code, read-only data and data.





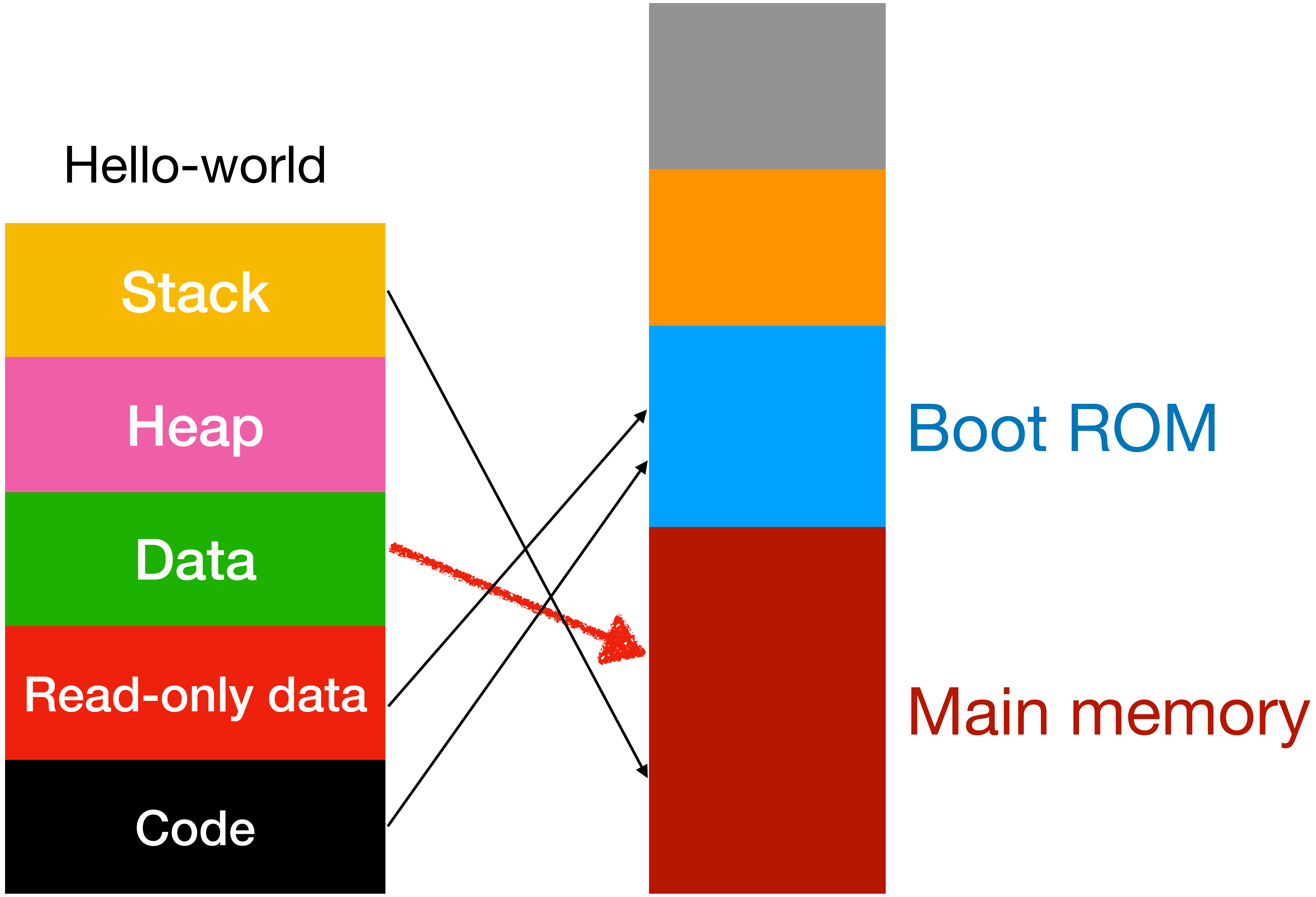
Boot up, step #1

- Boot ROM holds code, read-only data and data.
- The first few instructions setup the stack pointer.



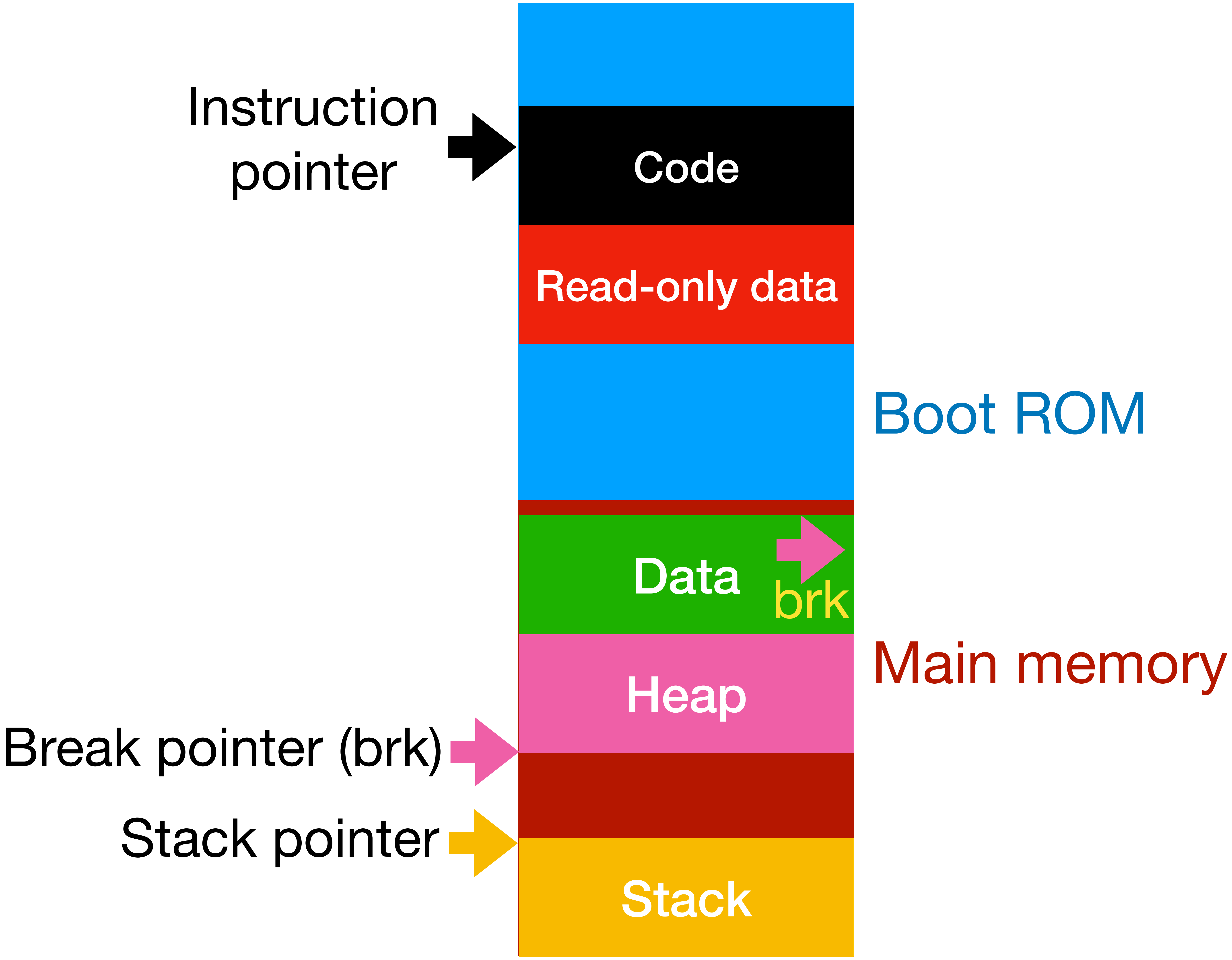
Boot up, step #2

- Boot ROM holds code, read-only data and data.
- The first few instructions setup the stack pointer.
- Copy data from ROM to main memory.



After the 2-step boot up

- Boot ROM holds code, read-only data and data.
- The first few instructions setup the stack pointer.
- Copy data from ROM to main memory.
- The break pointer is saved as 8-byte in data.



Reading the code

- Boot ROM holds code, read-only data and data.

  `earth/earth.lds`

- The first few instructions setup the stack pointer.

  `earth/earth.S`

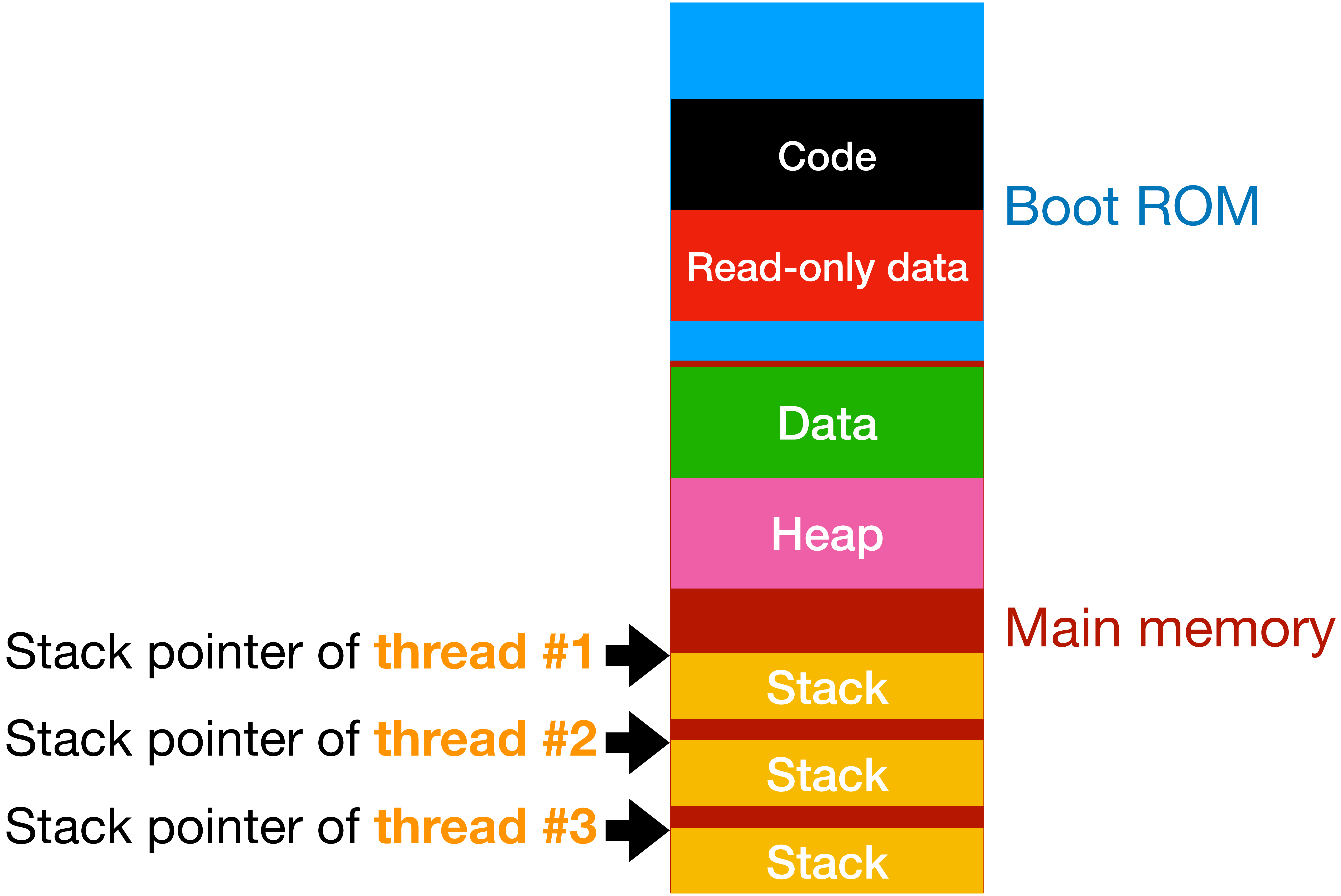
- Copy data from ROM to main memory.

  `earth/earth.c`
  `& first 2 loops in main()`

- The break pointer is saved as 8-byte in data.

  `earth/earth.lds`
  `& libc/malloc.c`

Hello-world → Multi-threading



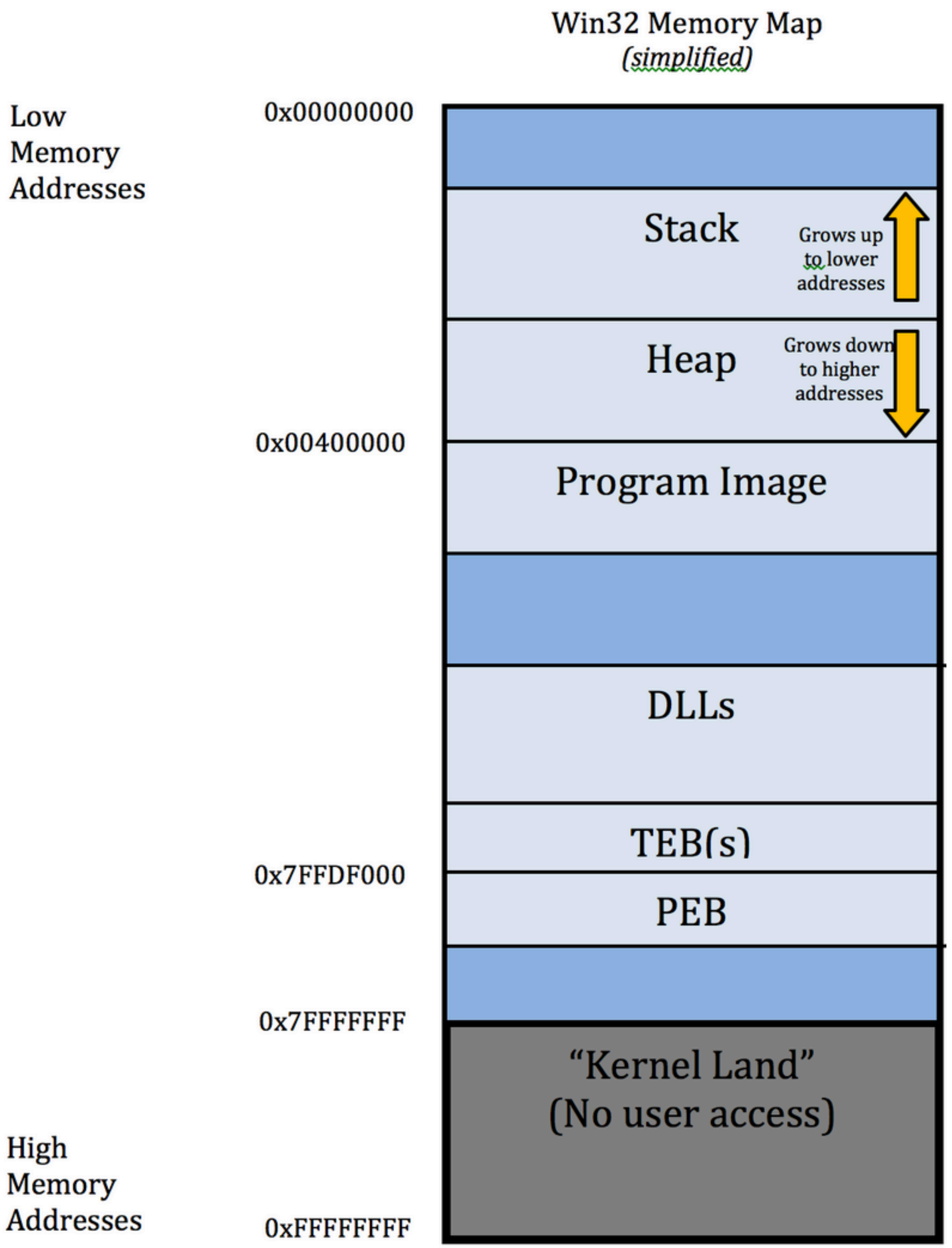
Memory Management

- Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  - Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory

Why translation?

- In P1, you write the code of every thread yourself.
- In an operating system,
  - Google writes the code of Chrome
  - Adobe writes the code of Photoshop
  - ...
- An operating system provides the standard memory layout specifying where to put code, stack, etc.

Standard memory layout of win32



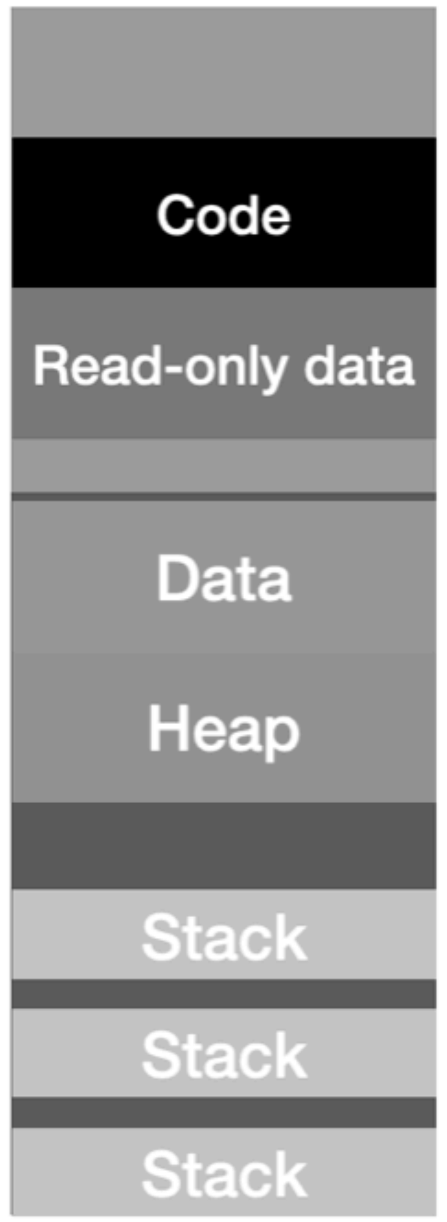
code, rodata and data of application

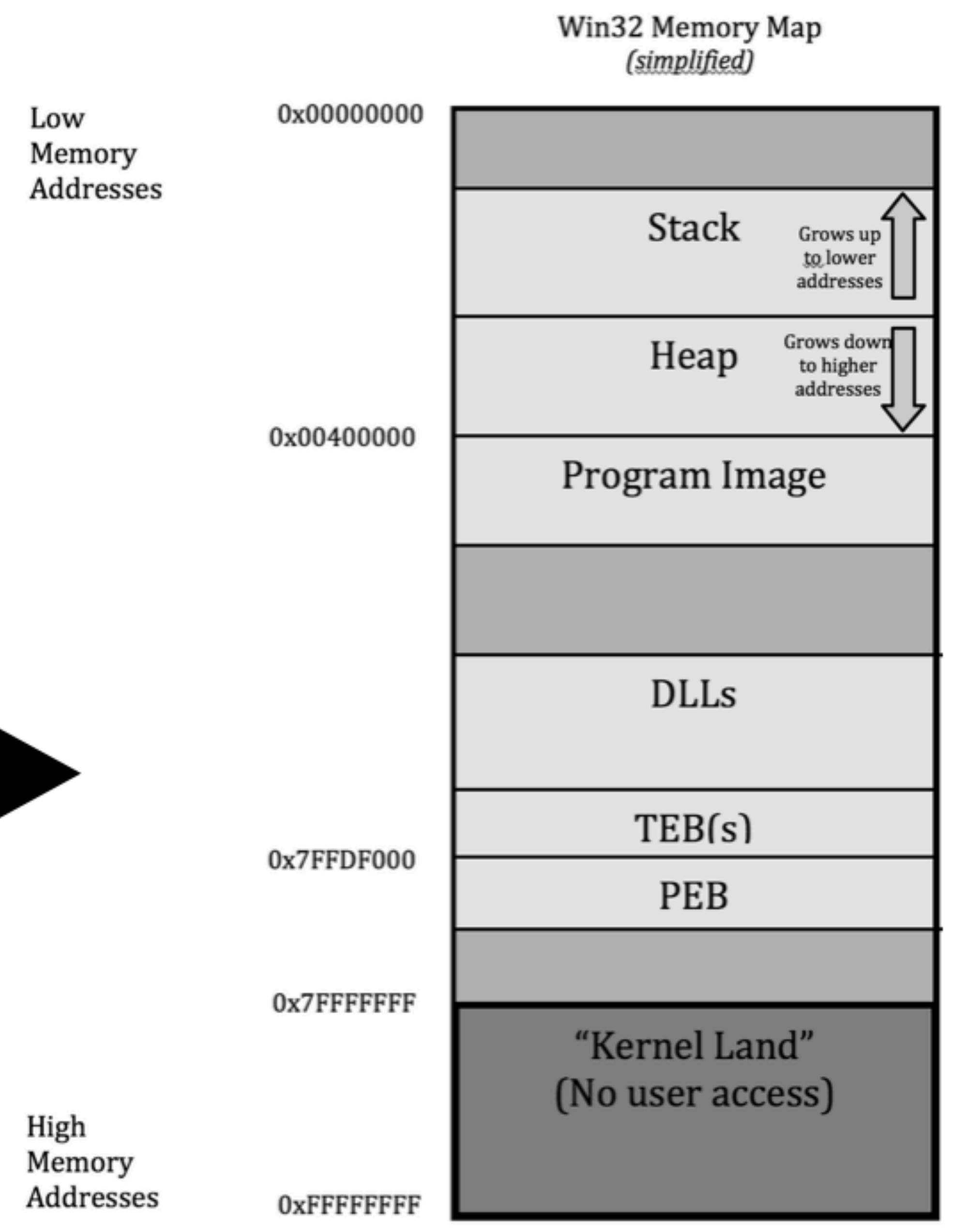
for dynamic library linking, out of today's scope

everything else, including the OS code and data

https://mikeczumak.com/blog/windows-exploit-development-part-1-basics/

Goal #1 of memory translation

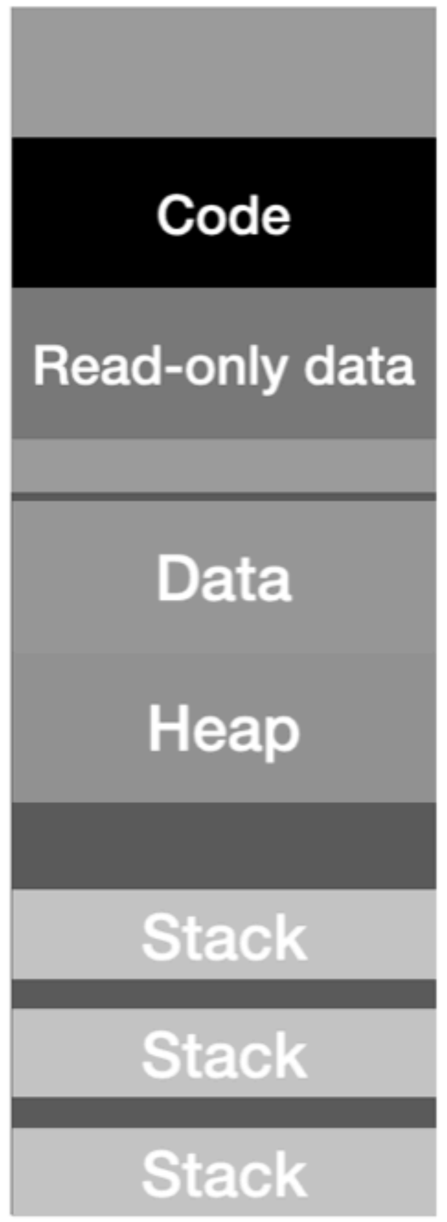


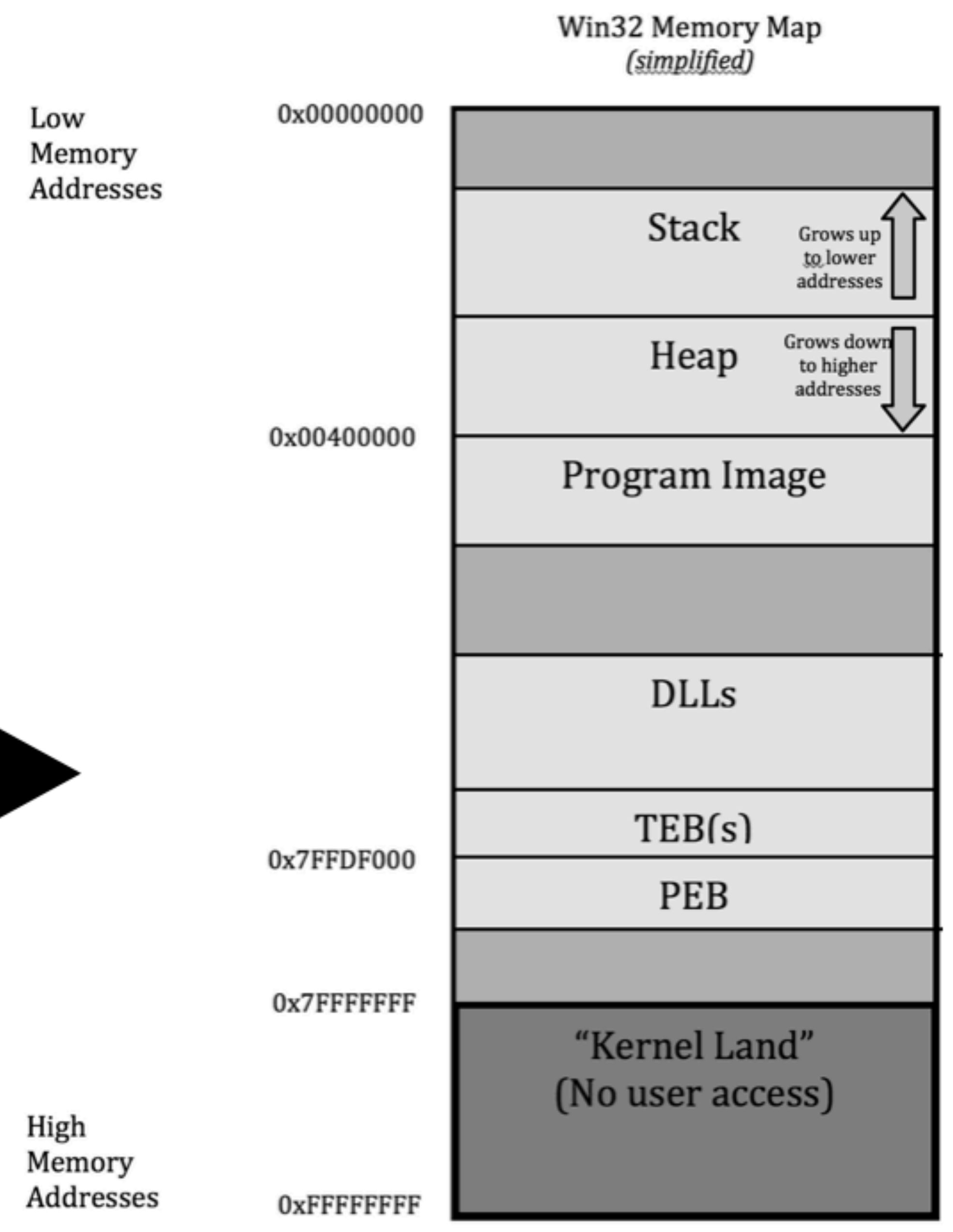


Different threads have different stack address, code address, etc.

Different processes have the same stack address, code address, etc.

Goal #2 of memory translation





Different threads share the same code, data, heap regions.

Different processes have separate code, data and heap regions.

Memory Management

- Hello World
- Why translation?
  - ➡ Case study: software TLB in egos
- Why protection?
  - Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory

Case study: Software TLB

- Every memory page is 4KB (0x1000 bytes).
- For every application in egos-2000:
  - 3 pages for code/rodata/data/heap
    - 0x0800_5000 … 0x0800_8000
  - 2 pages for stack
    - 0x8000_0000 … 0x8000_2000
- In addition, egos-2000 maintains a buffer of 256 pages (1 MB)

3 memory regions for Software TLB

Page * 3 **Application code/data/heap**

0x0800_5000

Page * 2 **Application stack**

0x8000_0000

Page * 256 **Memory buffer**

0x8000_4000

Operating system code/data/heap/stack is in other memory regions.
For example, consider your `thread_create()` and `thread_yield()` in P1.

RUNNING and RUNNABLE processes

Page * 3 code/data/heap of the RUNNING process

0x0800_5000

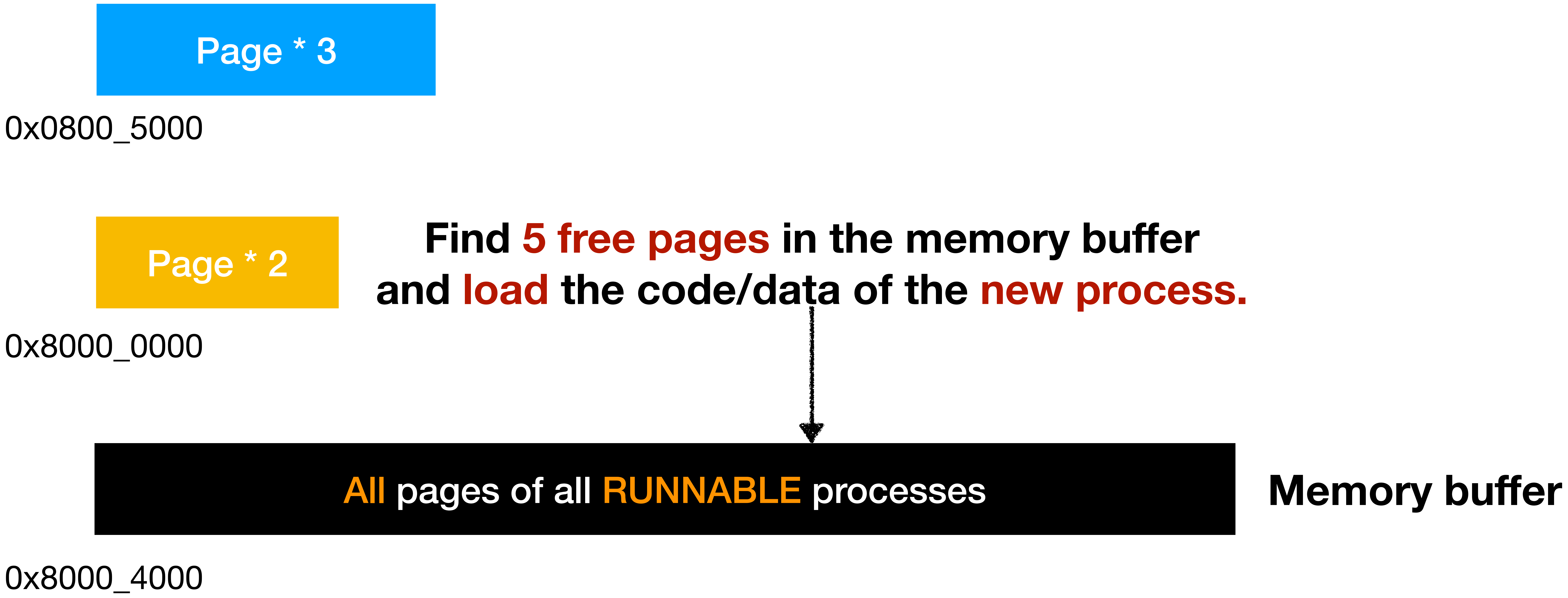
Page * 2 stack of the RUNNING process

0x8000_0000

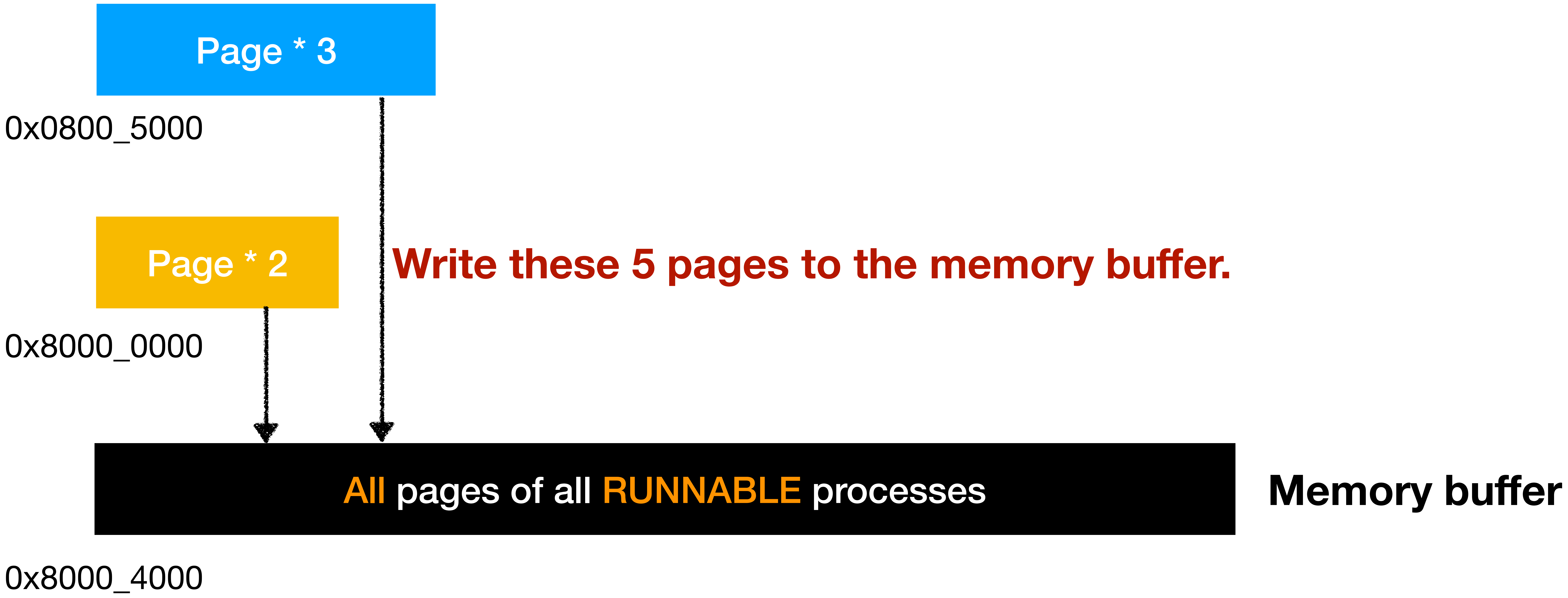
All pages of all RUNNABLE processes Memory buffer

0x8000_4000

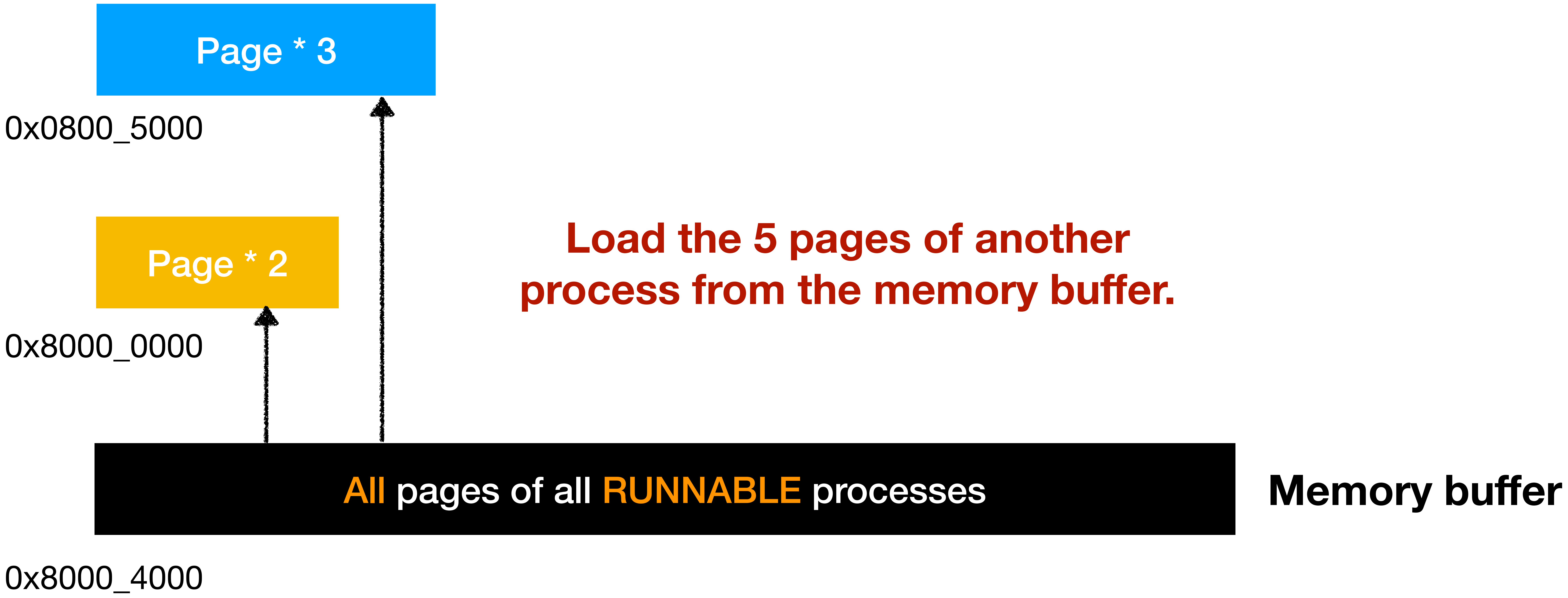
Additional step in `create()`



Additional step #1 in `yield()`



Additional step #2 in `yield()`



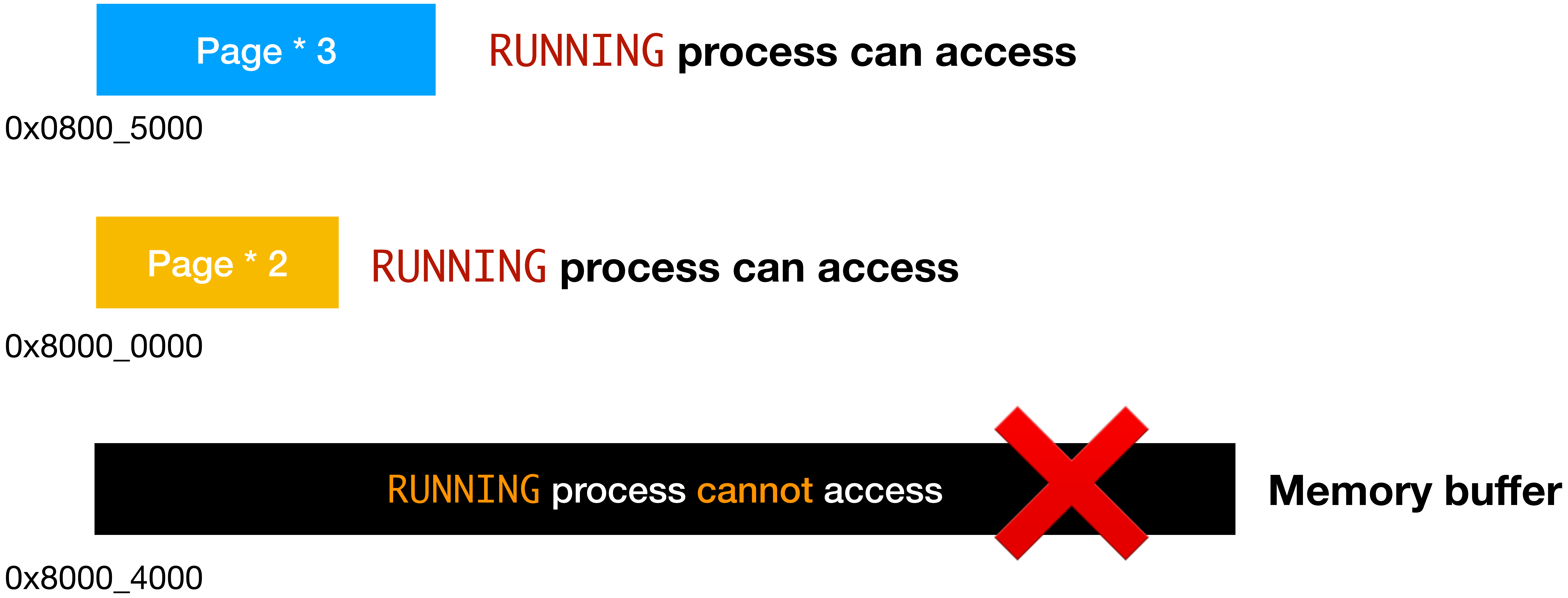
Software TLB summary

- Dedicated memory regions for user application.
  - 3 for code/data/heap + 2 for stack
- Comparing to the multi-threading in P1,
  - `create()` allocates and initializes memory pages for the code/data/heap of the process, in addition to stack
  - `yield()` moves memory pages between the dedicated regions (`RUNNING`) and the memory buffer (`RUNNABLE`)

Memory Management

- Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  - Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory

RUNNING process should not access the buffer



Introducing privilege levels

- Machine mode can access all memory regions.
- User mode can only access regions that are allowed by the machine mode.
    - Machine mode specify these regions and permissions.
    - Permissions are usually readable / writable / executable.

Machine mode specifies user permissions

Page * 3

code: r/-/x rodata: r/-/- data: r/w/- heap: r/w/-

0x0800_5000

Page * 2

stack: r/w/-

0x8000_0000

-/-/- (no access at all)

Memory buffer

0x8000_4000

Memory Management

- Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  - ➔ Case study: physical memory protection (PMP)
- Combining the two: page table and virtual memory

Physical memory protection (PMP)

- Read section 3.6 of the RISC-V manual
- There are 16 address CSRs and 4 config CSRs
  - `pmpaddr0` … `pmpaddr15 + pmpcfg0` … `pmpcfg3`
- For example, TOR means “top of region”:

```
/* Setup PMP TOR region 0x00000000 - 0x08008000 as r/w/x */
asm("csrw pmpaddr0, %0" :: "r" (0x08008000));
asm("csrw pmpcfg0, %0" :: "r" (0xF));
```

Memory Management

- Hello World
- Why translation?
  - Case study: software TLB in egos
- Why protection?
  - Case study: physical memory protection (PMP)

➔ Combining the two: page table and virtual memory

Page table and virtual memory

- Achieve the same goals as PMP + software TLB.
- In P3, page table translation is left to you as an open-ended hobby project, not graded.
  - Again, the goal of 4411 is to have fun.
- There are 29 lines of code setting up some example page tables in egos-2000. See the handout for details.

Homework

- P3 will be due on Nov 4. You will implement
  - system call, memory protection and exception handling
- Next lecture: I/O bus and device driver