Preliminaries

CIS531 — Fall 2025, Syracuse

Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

Preliminary Roadmap

We will cover a few things today:

- Lexical analysis (lexing)
- Grammars and parsing
- Assembly / machine organization crash course

Grammars

- In this class, we'll often need to describe **languages**
- We specify the *definition* of a language using a **grammar** consisting of **terminals** (terms which match *exactly*), *and* **nonterminals** (recursive matching)

- Here's an example grammar:

```
P ::= (print e)
e ::= Number
    | e * e
    | e + e
```

The terminals are *, +, print, (, and); the nonterminals are P, e, and Number
We elide the definition of Number (any regex can be written in grammar form)

Lexical Analysis

- The **terminals** in the grammar represent atomic tokens
- Traditionally, we separate syntactic analysis into two phases:
  - **Lexical analysis**, which recognizes individual tokens from the input byte stream and outputs logically-related chunks
  - **Parsing**, which takes these tokens as input (they the ***terminals*** of the grammar) and produces a syntax / parse tree
- The "lexer" is typically written in terms of *regular expressions*:
  - Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - The "|" represents logical disjunction—a digit is one of those individual 0-9
  - Number = -?Digit+
    - Possibly a leading - (the ? means "either zero or one" of the preceding thing)
    - At least one digit
- ***We will put this off until a bit later on in the course!***

Parsing

- Parsing is the act of taking an input string and matching it against the grammar:

```
P ::= (print e)
e ::= Number
    | e * e
    | e + e
```

Which of the following can be generated by the grammar P:

- [ ] (print 5)
- [ ] (print (5 * (2 + 3)))
- [ ] (print (3 * (print 2)))

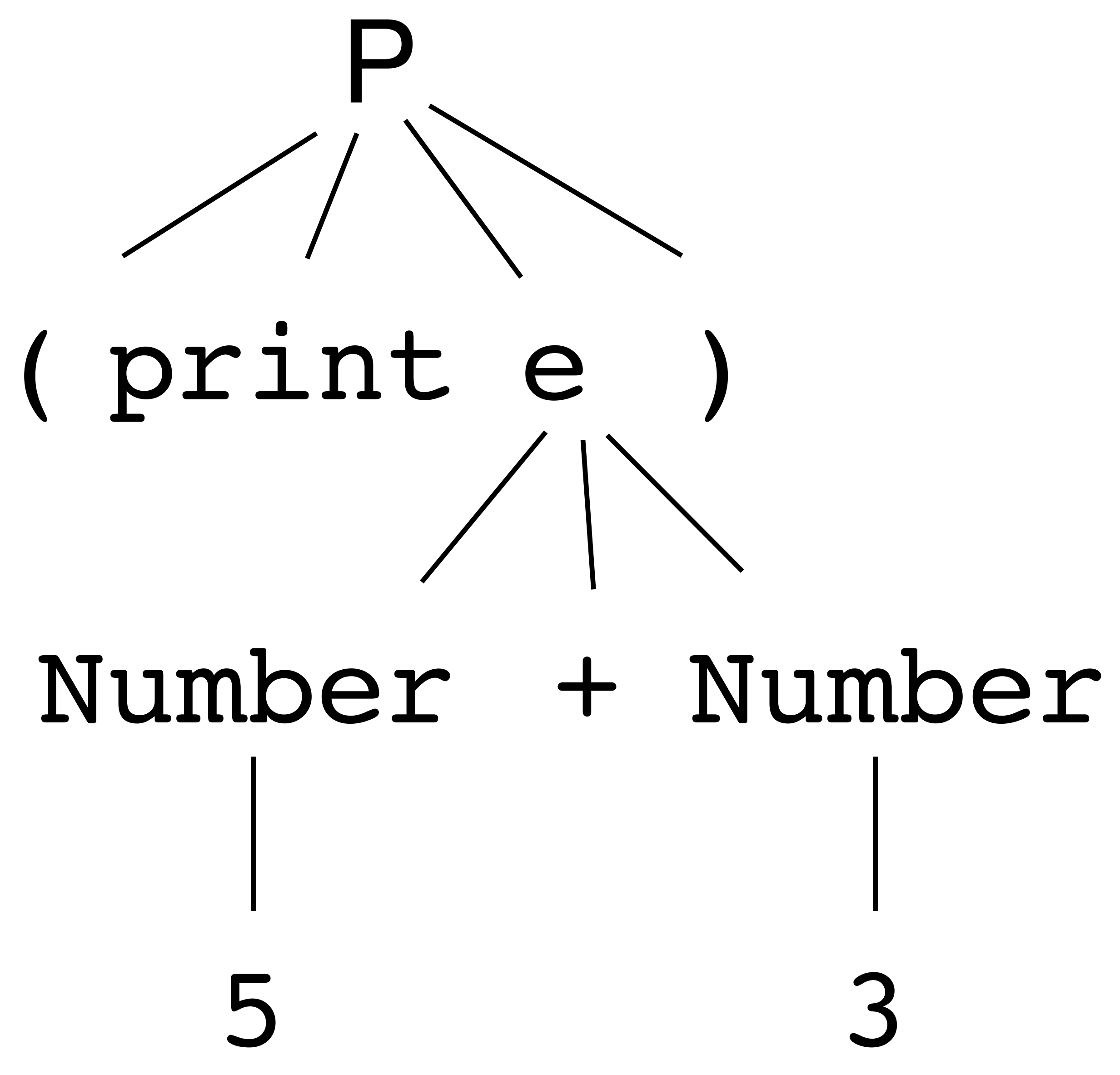
Answer: first two

Parsing…

Token stream
(Output of lexer)

Parse!

`(print (5 + 3))` →




```
P ::= (print e)
e ::= Number
    | e * e
    | e + e
```

- In practice, we don't just care about *whether* a string matches the grammar
- What we want is to a data structure that tells us **how** the string matched
- Generally, we want this in the form of a ***tree***

x86_64 (AT&T/GAS/Clang-style) Assembler

- To build a compiler, you have to learn (some amount of) Assembly code
- We will target **AT&T-style x86_64** as used in GAS, clang/LLVM, and similar tools
  - This is a very common assembly variant you will often see in practice
  - Don't worry if you have a Mac with an ARM chip—our infrastructure will handle it

<BEGIN>
Crash Course on x86_64 assembly!

- ***Required*** reading:
  - http://ian.seyler.me/easy_x86-64/
  - https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
  - http://nickdesaulniers.github.io/blog/2014/04/18/lets-write-some-x86-64/
- Optional (but strongly encouraged)
  - https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
  - https://www3.nd.edu/~dthain/courses/cse40243/fall2015/intel-intro.html

The basics...

- Assembly code consists of **instruction sequences**
  - Grouped into “functions” (procedures)

- Each instruction does a very simple task (add, multiply, jump)

- There are a limited number of variables (registers)
  - x86-64 has 16 of these! 2 hold pointers to stack (rsp/rbp)

- If you need more memory (e.g., for storing an array), must store in stack / heap / etc...

x86_64 is huge, but you only need to know a little bit...

- Ugly instruction set: decades of development and extensions
- In practice, you can deal with a very small subset of x86_64
- We will write code to generate the boilerplate
- For a while, we will focus on compiling only a single function

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

This is not **code**, you are telling the processor to put some **data** somewhere and name it _hello

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Commands starting with dots (.) are directives that tell the assembler how to lay out your program

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

.text says “put this in the text segment”

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

.globl means “make this global”

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

OS **assumes** you will have a function named _main

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

The %rsp variable (register) is a 64-bit pointer to the stack.

Remember, the stack grows **down**

First command **subtracts 8** from %rsp

This "allocates" 8 bytes on the stack, so that our program can store data there.

This is complicated! More on it later

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Moves 0 into %rax (general purpose 8-bit register)

Note:
opcode, source, dest

(This is the convention in AT&T syntax)

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Loads address of `_hello` into rdi

`printf` is a special “variadic” function, so # extra arguments has to be put into rax

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Actually performs the function call!

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movq $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Returns here!

Since we want to call exit(0), need to put 0 in %rdi

Cross-compiling on a Mac (M1/2/3/...)

- We use clang's *cross-compiler* to build a Mach-O x86_64 object file
- Then, we use clang to link the object (.o) file into a Mach-O executable
- Finally, we run it: Rosetta loads it and translates it on-the-fly to run on ARM!

```
kkmicins@laptop ∂ → clang -target x86_64-apple-darwin -c ex0.s
kkmicins@laptop ∂ → clang -target x86_64-apple-darwin ex0.o -o prog
kkmicins@laptop ∂ → ./prog
Hello, world!
```

Registers: very fast, on-chip data

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

Base pointer

(Start of frame)

Stack pointer

(Top of stack)

Registers: very fast, on-chip data

- Also some special registers you can’t modify directly: instruction pointer, flags
- Registers are a precious commodity: you want to store local variables in registers
  - Historically: ensuring something certain values kept in registers was a reason to write manually-tuned Assembly code (not as common in modern times)

Preview: register allocation (several weeks from now…)

- You will eventually **run out** of registers, at which point you **have** to “spill” to the stack
  - Stack stores local values: medium-sized chunk of memory that you can push/pop
  - Push on function entry, pop on function exit—local values disappear once function done

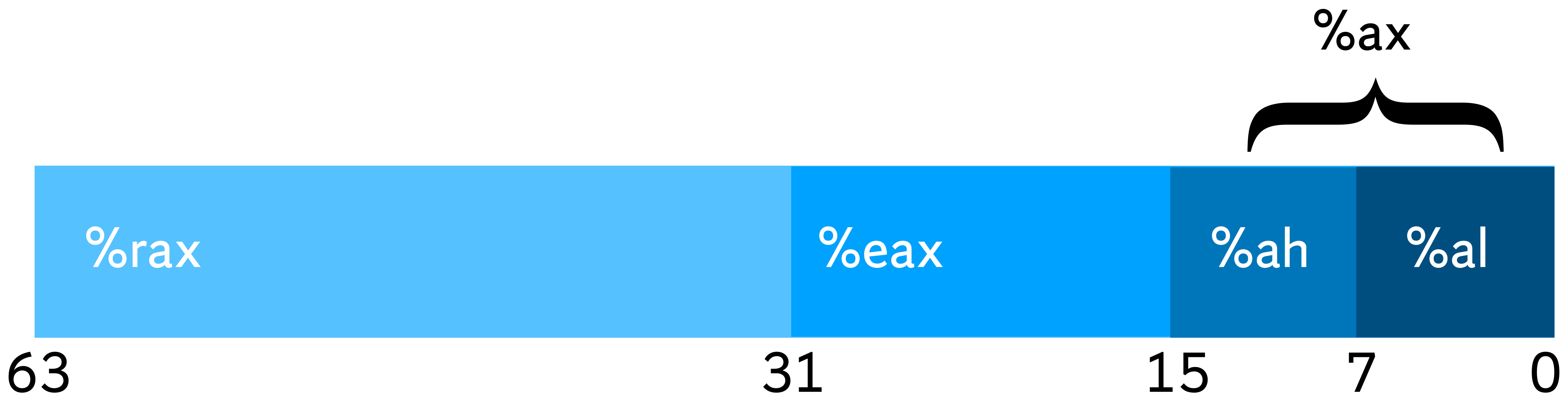
Example: Register Allocation (yourself)

- Consider the following code. If you only have 3 registers, can you keep every variable in a register?

```
int foo(int arg1, int arg2) {
  x = arg1;
  y = arg2;
  z = 0;
  if (x > y) {
    a = y;
    z = a+1;
    print(z);
  } else {
    b = x;
    z = x + b;
    print(z);
  }
}
```

Smaller registers are *nested* inside bigger ones

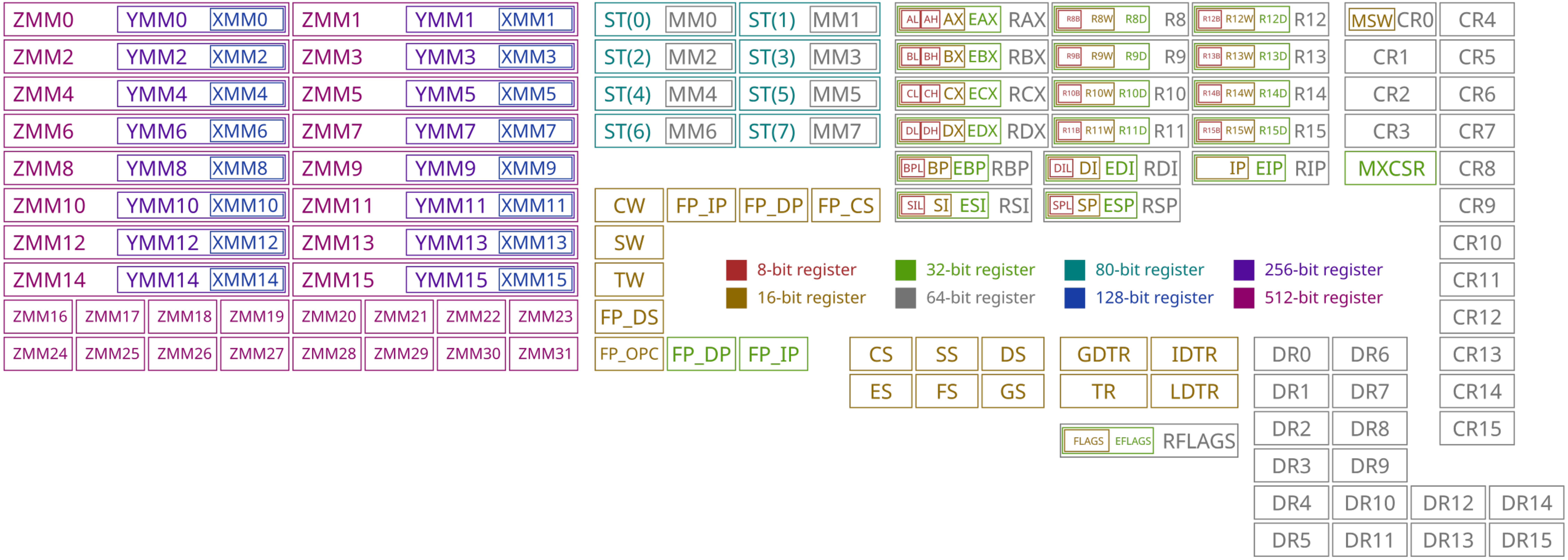
- %rax is one of the most common general-purpose registers
- But, if you use %eax, you're **really** using the ***lower 32 bits*** of %rax
- In this class, we'll make it easy: everything is 64 bits!
  - Changing this is an interesting exercise, potential final project?



Modern x86: x86_64

- Today we mostly have 64-bit ISAs
- 32-bit registers still used (e.g., int in C++ on my machine)
- 16-bit basically gone, 8-bit totally gone
  - Some code out there that still works with the 32/16/8 bit registers, though
- Assembly instructions have **suffixes**: denote which bit-width they're working on
  - E.g., movq says "move into a **quadword** (8-bytes)"
  - movl says "move into a **long** (4-bytes)"
  - We will try to keep things simple by working with 64-bit values values!

x86_64 register map



https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg

To represent `0x1234567890abcdef`

12	34	56	78	90	ab	cd	ef

Most Significant Byte   Least Significant Byte

Endianness

- x86 is a **little-endian** architecture
- If an n-byte value is stored at addresses a to a+(n-1) in memory, byte a will hold the least significant byte
- Note: values ***within*** bytes *do not* change—only the **byte order** changes
- How would 0x1234567890abcdef be stored in memory on a big/little-endian machine, starting at address 0xAA00?

Instruction types

- Moving memory around: mov, load, store, etc…
- Arithmetic: add, mul, div, shift, etc…
- Tests: (in)equality, comparison (less/greater than), etc.
- Branch: conditional or unconditional
- Call: direct (target hardcoded into the call) or indirect (function being called isn't known until runtime—costlier to implement, harder to analyze!)
- Floating point, matrix instructions, etc: won't talk a ton about these!
- More esoteric instructions: atomics, cache-bypassing instructions, …

Addressing modes

- Instructions take *operands* as arguments, and these operands may not always be allowed to be (for example) immediate values.
  - For example, you can't add two immediate values, have to mov in a register first, followed by add of immediate to register
  - Question: why enforce this restriction?
  - Answer (several): encoding space of instructions is precious—don't waste space on redundant computation (why would you not just manually do the calculation?), among others…

Addressing modes (register)

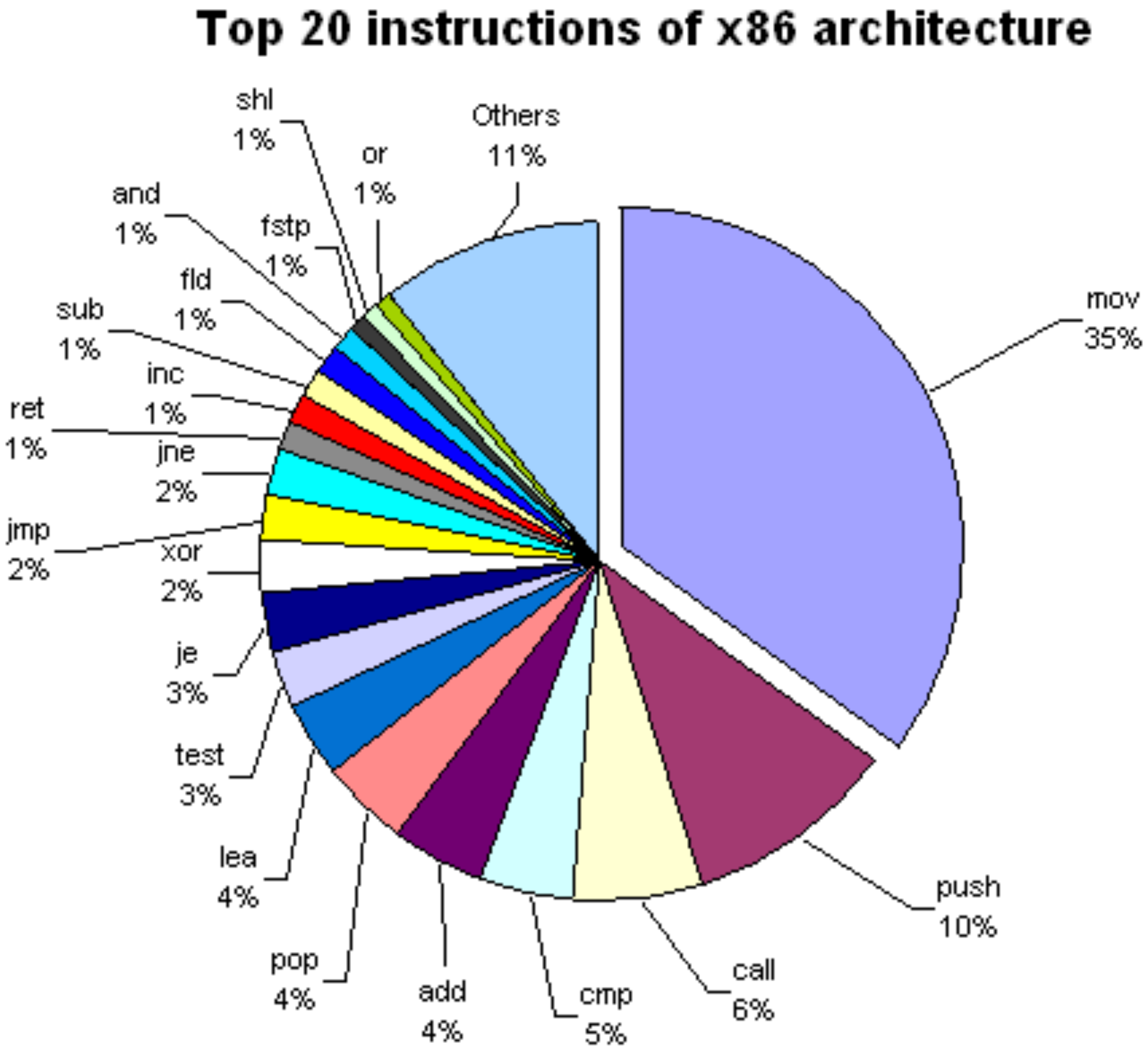
“Move the value from register rax into the register rbx”

Opcode name

Destination

```
mov  %rax, %rbx
```

Source



Plurality of instructions are **mov**s

Then **push**

Then **call**

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Save %rbp onto the stack (need to do this for alignment)

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move 28 into rbx

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move 23 into rax

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Add rax to rbx, store result in rax

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

“Sign extend” %rax into %rdx:%rax

(The idivq instructions expects its arguments to be in **both** rdx and rax! So must sign extend!)

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move 2 into rbx

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Integer divide %rdx:%rax by %rbx

(Since %rdx will be 0 here, this basically means: %rax/%rbx, store result in %rax, remainder stored in %rdx)

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move result into %rdi in preparation to call `exit`

Quiz: What would equiv C++ code look like?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Many other solns possible!

```
int x = 28;
int y = 23;
x += y;
x /= 2;
exit(x);
```

Example: finding max of two ints

```
.text
.globl _main
_main:
  pushq %rbp
  movq $8, %rax
  movq $7, %rbx
  cmp %rax, %rbx
  jg _mov_needed
  jmp _no_mov_needed
_mov_needed:
  movq %rbx, %rax
_no_mov_needed:
  movq %rax, %rdi
  call _exit
```

Push %rbp

Move 8 into rax

Move 7 into rbx

Compare rax and rbx

If %rax is greater, go to _mov_needed

Unconditional jump to _no_mov_needed

Label for _mov_needed

Join point for computation

Move rax into rdi to call _exit

Quiz: what would C++ code look like?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $8, %rax
  movq $7, %rbx
  cmp %rax, %rbx
  jg _mov_needed
  jmp _no_mov_needed
_mov_needed:
  movq %rbx, %rax
_no_mov_needed:
  movq %rax, %rdi
  call _exit
```

Quiz: what would C++ code look like?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $8, %rax
  movq $7, %rbx
  cmp %rax, %rbx
  jg _mov_needed
  jmp _no_mov_needed
_mov_needed:
  movq %rbx, %rax
_no_mov_needed:
  movq %rax, %rdi
  call _exit
```

```
int x = 8; // rax
int y = 7; // rbx
if (y > x)
  x = y;
exit(x);
```

Example: using a loop

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

%rax will accumulate a value

%rbx is going to track when to exit

%rcx will count up by one until it hits %rbx

_ starts a **label**

Compare %rcx and %rbx

If previous comparison was =, jump to …

Unconditional jump to _loop_begin

Target of jump after cmp

Move %rax into %rdi to call _exit

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

Quiz: what does this program compute?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

Quiz: write corresponding C++ code for this

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

```
int x = 0; // rax
int y = 6; // rbx
int z = 0; // rcx
while (y != z) {
  x += x;
  ++z;
}
exit(x);
```

Memory: a **giant chunk of bytes**

You can read from it and write to it in 1/2/4/8/16-byte increments

```
mov  (%rax), %rbx
```

“Move the value **at address** %rax into register %rbx”

Opcode name

Destination

```
mov  (%rax), %rbx
```

Source

%rax	0xffffffff00000000	0xffffffff00000008	0xaf23c8a223356ac
		0xffffffff00000000	0xdeadbeefdeadbeef
%rbx	0x1234123412341234		

“Move the value **at address** %rax into register %rbx”

Opcode name

Destination

`mov (%rax), %rbx`

Source

%rax	0xffffffff00000000	0xffffffff00000008	0xaf23c8a223356ac
		0xffffffff00000000	0xdeadbeefdeadbeef
%rbx	0xdeadbeefdeadbeef		

“Move the value **at address** %rax+8 into register %rbx”

Opcode name

Destination

```
mov   8(%rax), %rbx
```

Source

%rax 0xffffffff00000000

0xffffffff00000008 0xaf23c8a223356ac

0xffffffff00000000 0xdeadbeefdeadbeef

%rbx 0xaf23c8a223356ac

A few other more complicated ones that allow
you to add registers, offsets, etc…

Different instructions allow different addressing-modes

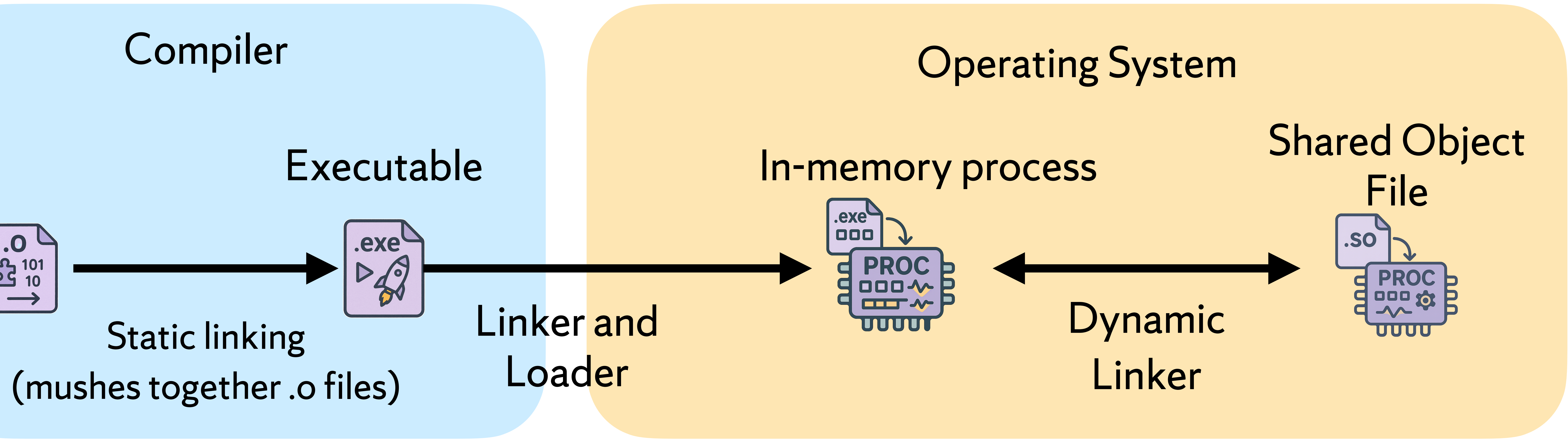
```
# Full example: load *(ebp + (edx * 4) - 8) into eax
movq    -8(%ebp, %edx, 4), %eax
# Typical example: load a stack variable into eax
movq    -4(%ebp), %eax
# No index: copy the target of a pointer into a register
movq    (%ecx), %edx
# Arithmetic: multiply eax by 4 and add 8
leaq    8(,%eax,4), %eax
# Arithmetic: multiply eax by 2 and add edx
leaq    (%edx,%eax,2), %eax
```

```
# Full example: load *(ebp + (edx * 4) - 8) into eax
movq    -8(%ebp, %edx, 4), %eax
# Typical example: load a stack variable into eax
movq    -4(%ebp), %eax
# No index: copy the target of a pointer into a register
movq    (%ecx), %edx
# Arithmetic: multiply eax by 4 and add 8
leaq    8(,%eax,4), %eax
# Arithmetic: multiply eax by 2 and add edx
leaq    (%edx,%eax,2), %eax
```

`leaq` is "load effective address (quad)." You can think of it as the assembly analogue of C++'s & (address of) operator

What happens *after* the executable is generated

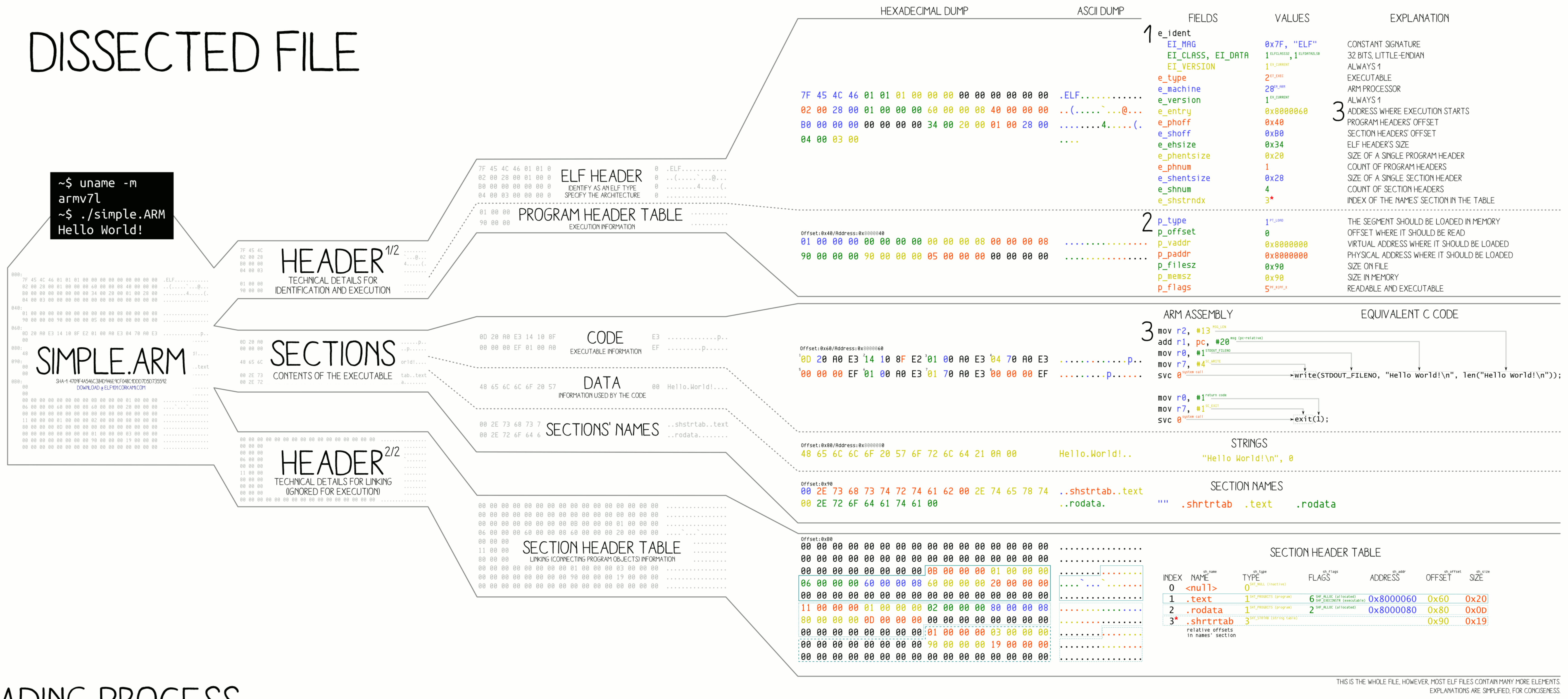
- The executable is written in a very specific format which the OS understands
  - Examples: Windows Portable Executables (PE .exes), Linux ELF, Mac Mach-O, other lesser-known
- The OS typically organizes process memory into ***segments***
- The loader (roughly: what happens after execve) knows how to populate the processes memory from the .exe file. The loader and dynamic linker collaborate to load the .exe into memory
- At runtime, some calls to externally-defined functions may need to be resolved by the dynamic linker—this is accomplished by some technical systems tricks which I will not explain in detail now



ELF101 a Linux executable walkthrough

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE



LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED

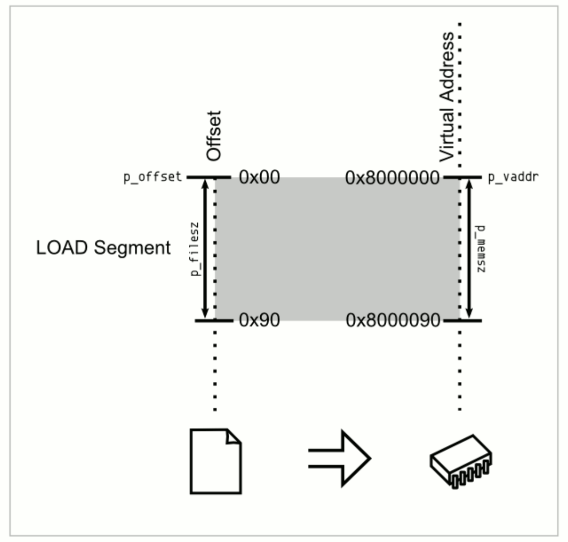
THE PROGRAM HEADER IS PARSED

(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY

ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED

SYSCALLS ARE ACCESSED VIA:

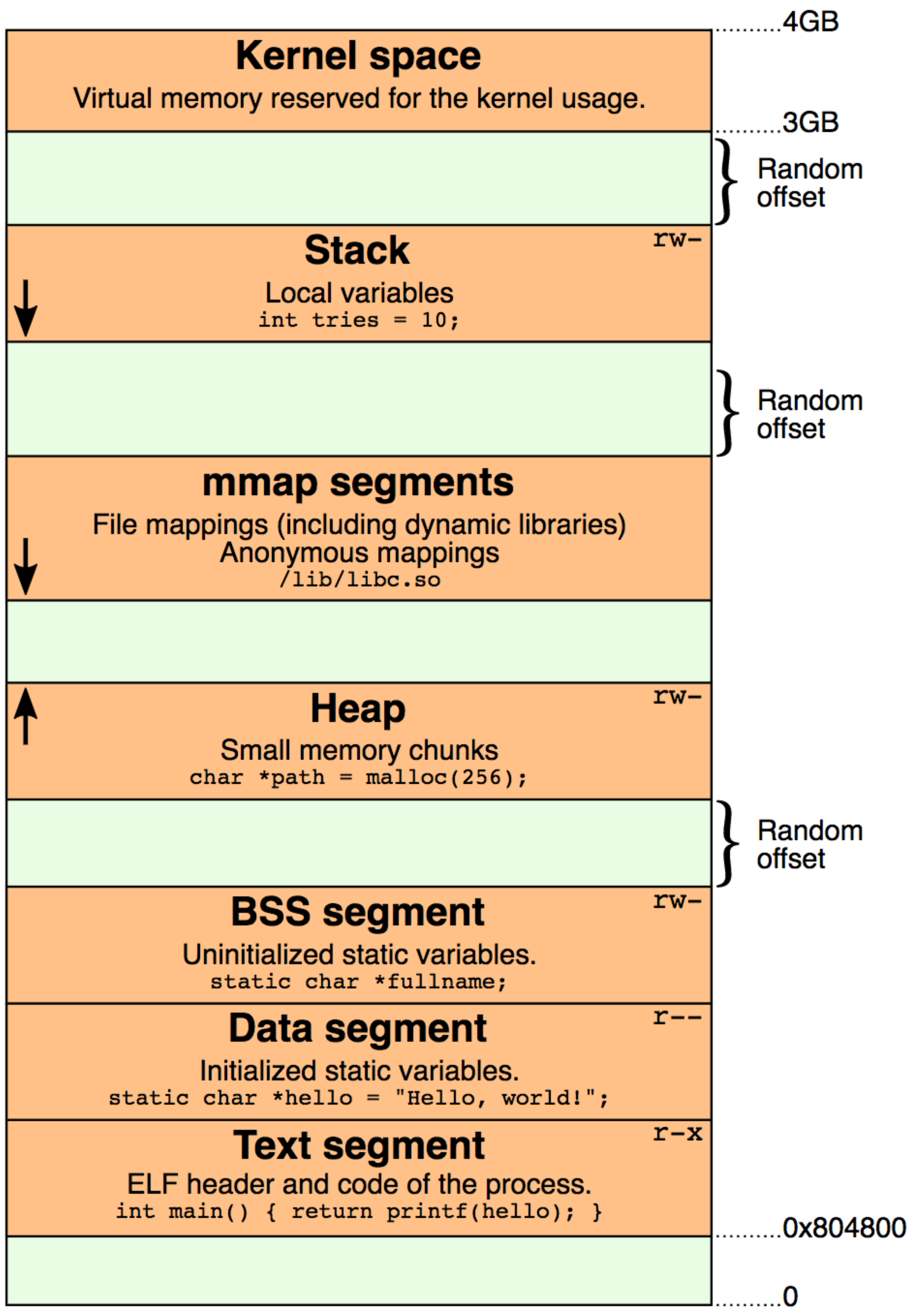
- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I.
FOR UNIX SYSTEM V, IN 1989

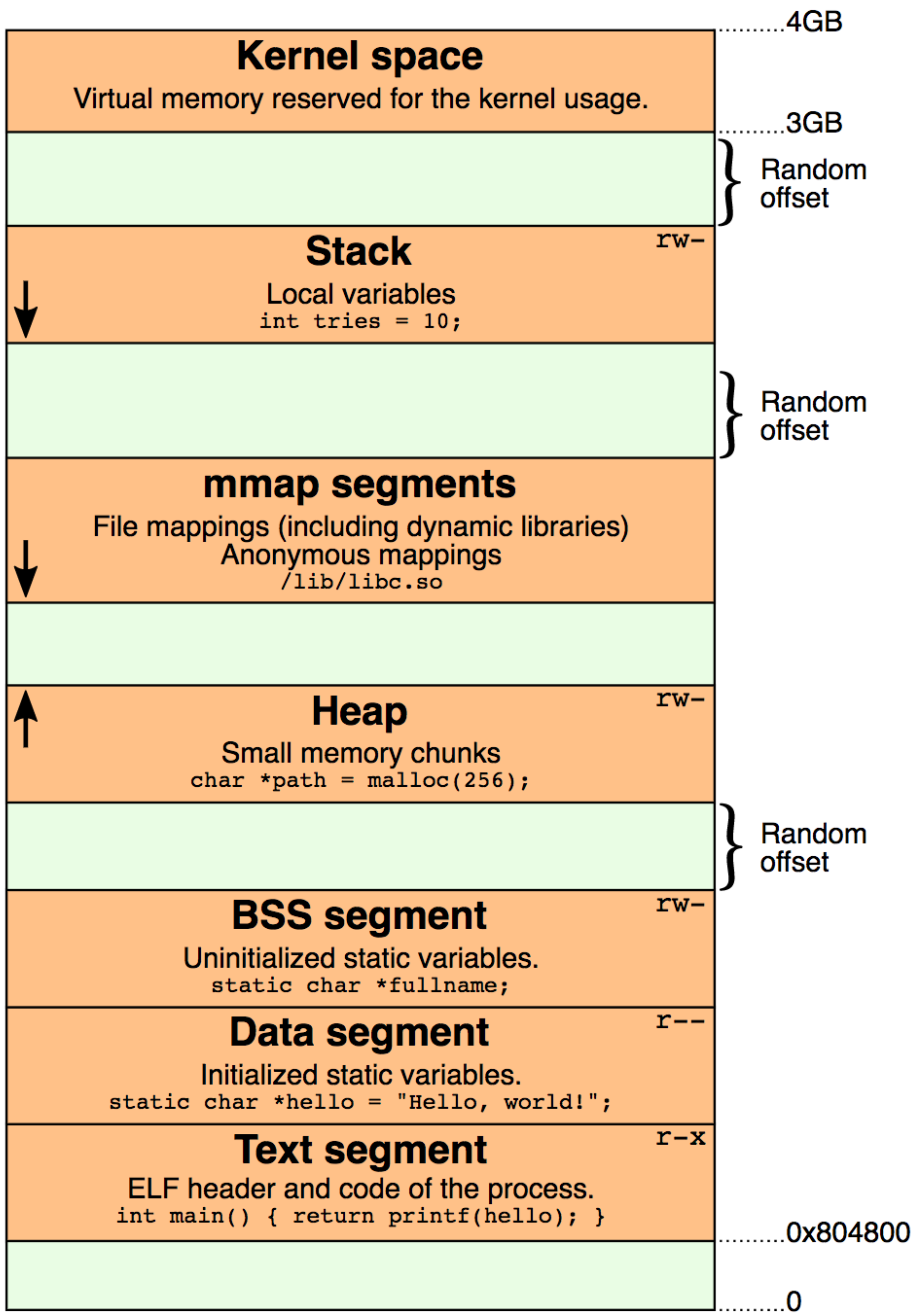
THE ELF IS USED, AMONG OTHERS, IN:

- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS



Kernel memory

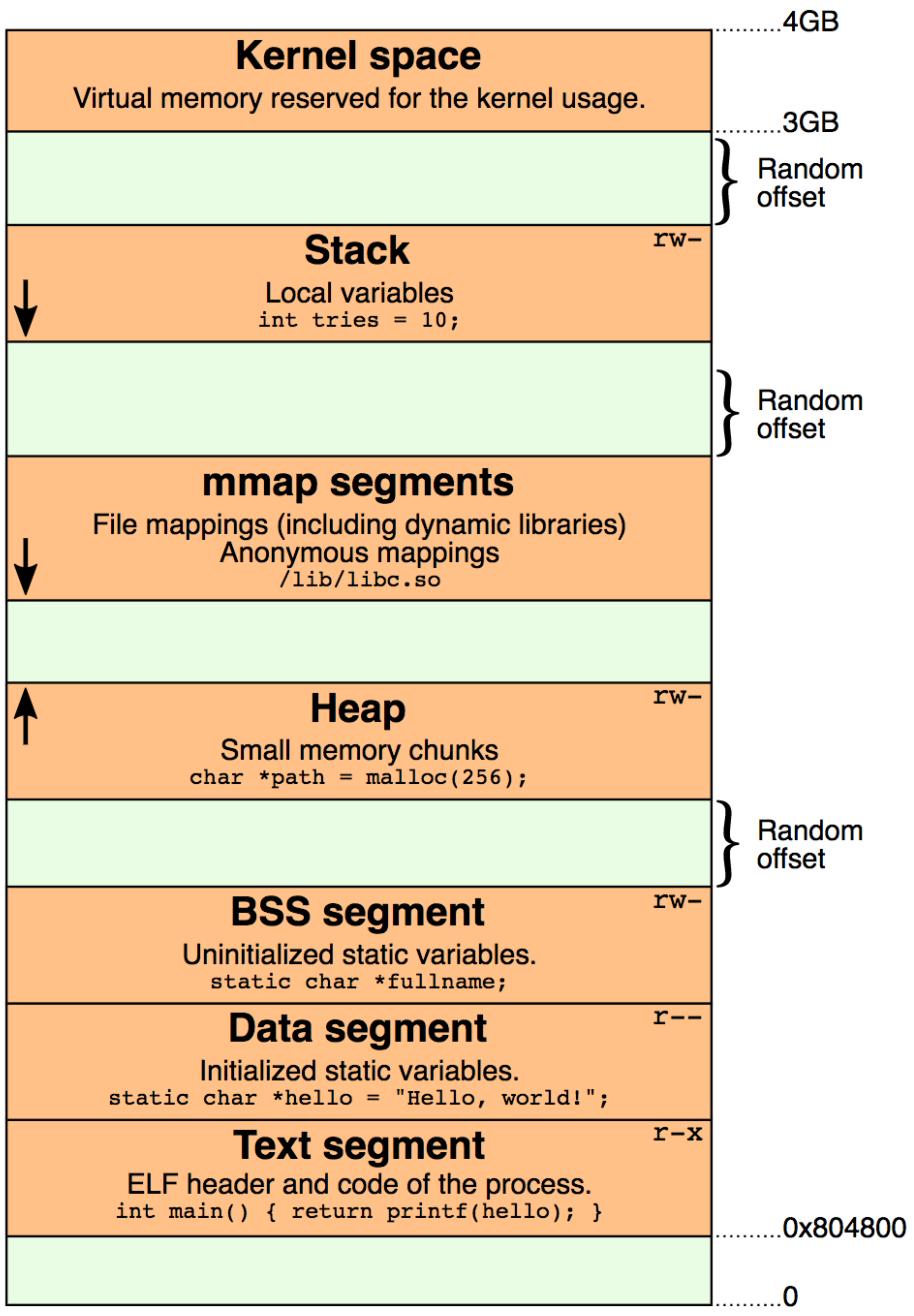
Your OS uses it



Stack: push / pop

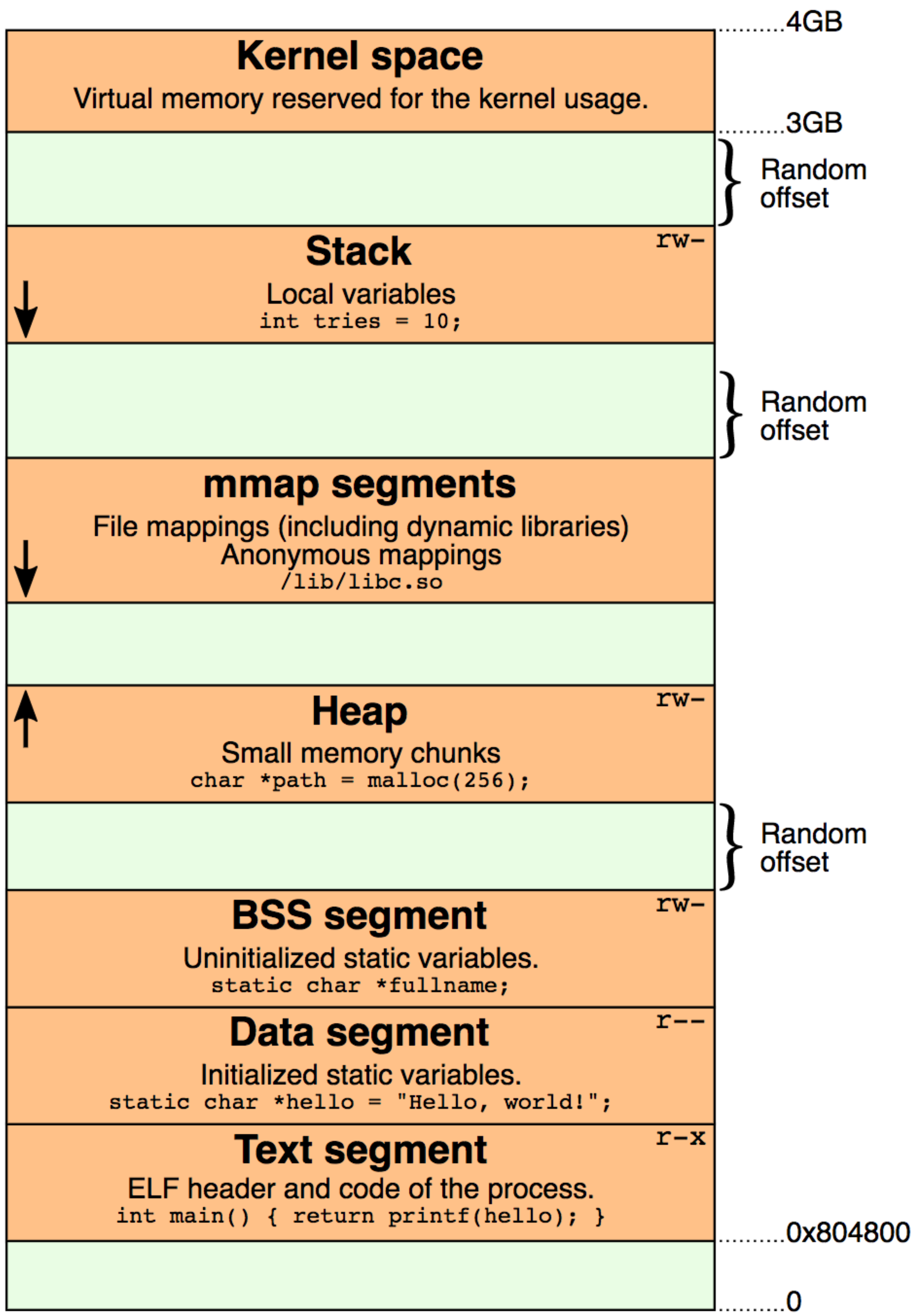
Very important:

The stack grows **down**



mmap segments

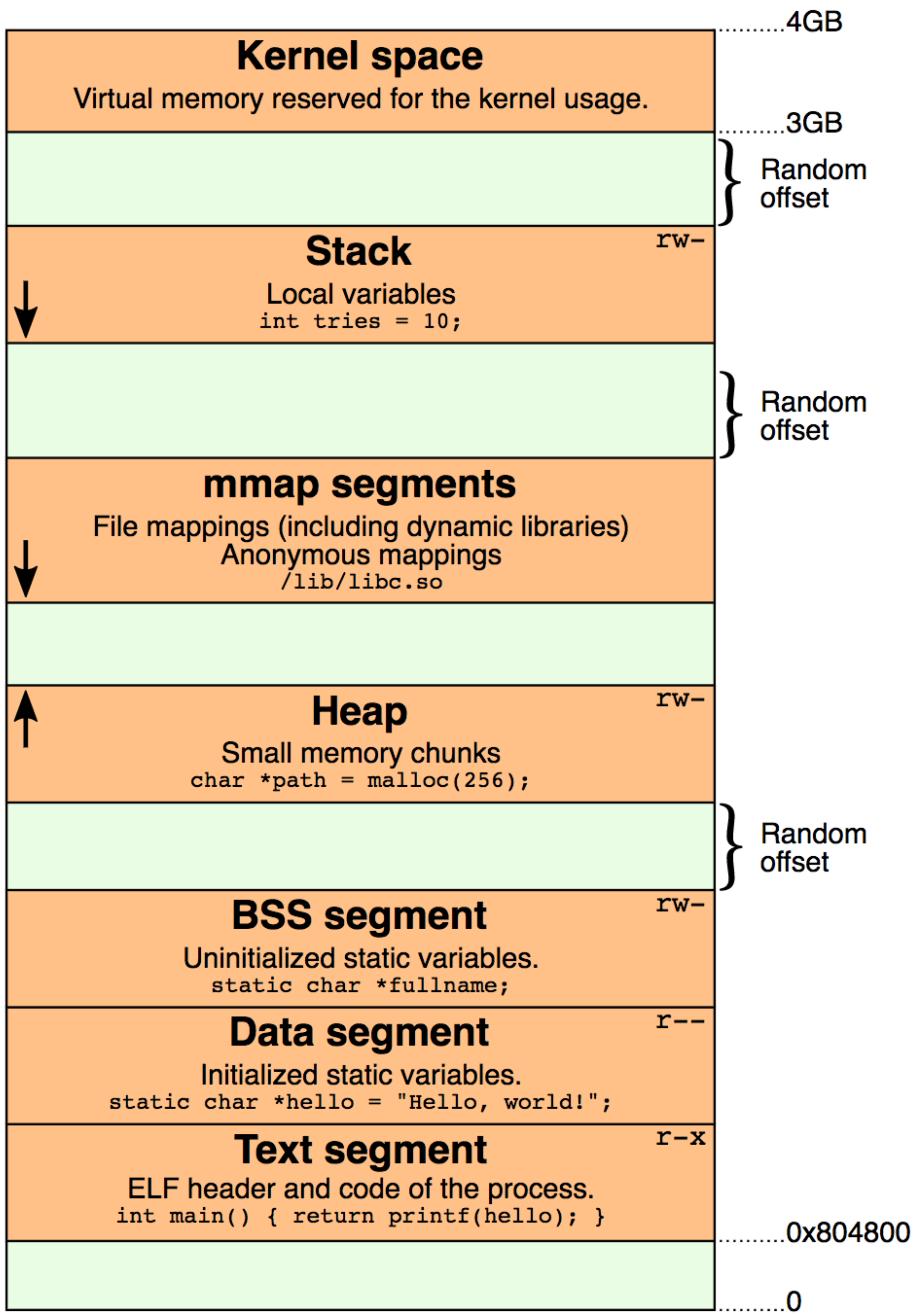
Allows you to **map** a file to memory



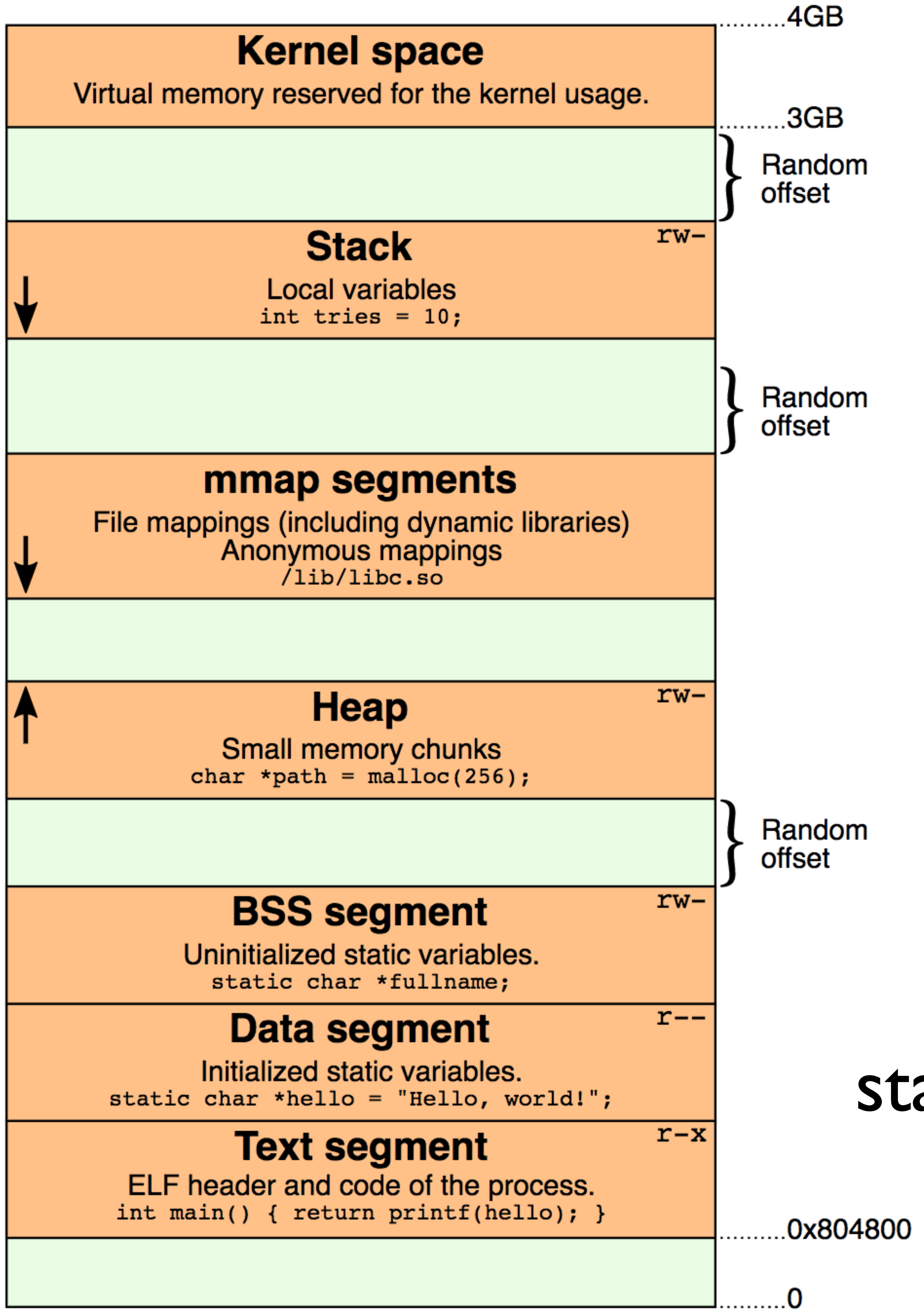
Heap: dynamic allocation

C++: New / delete

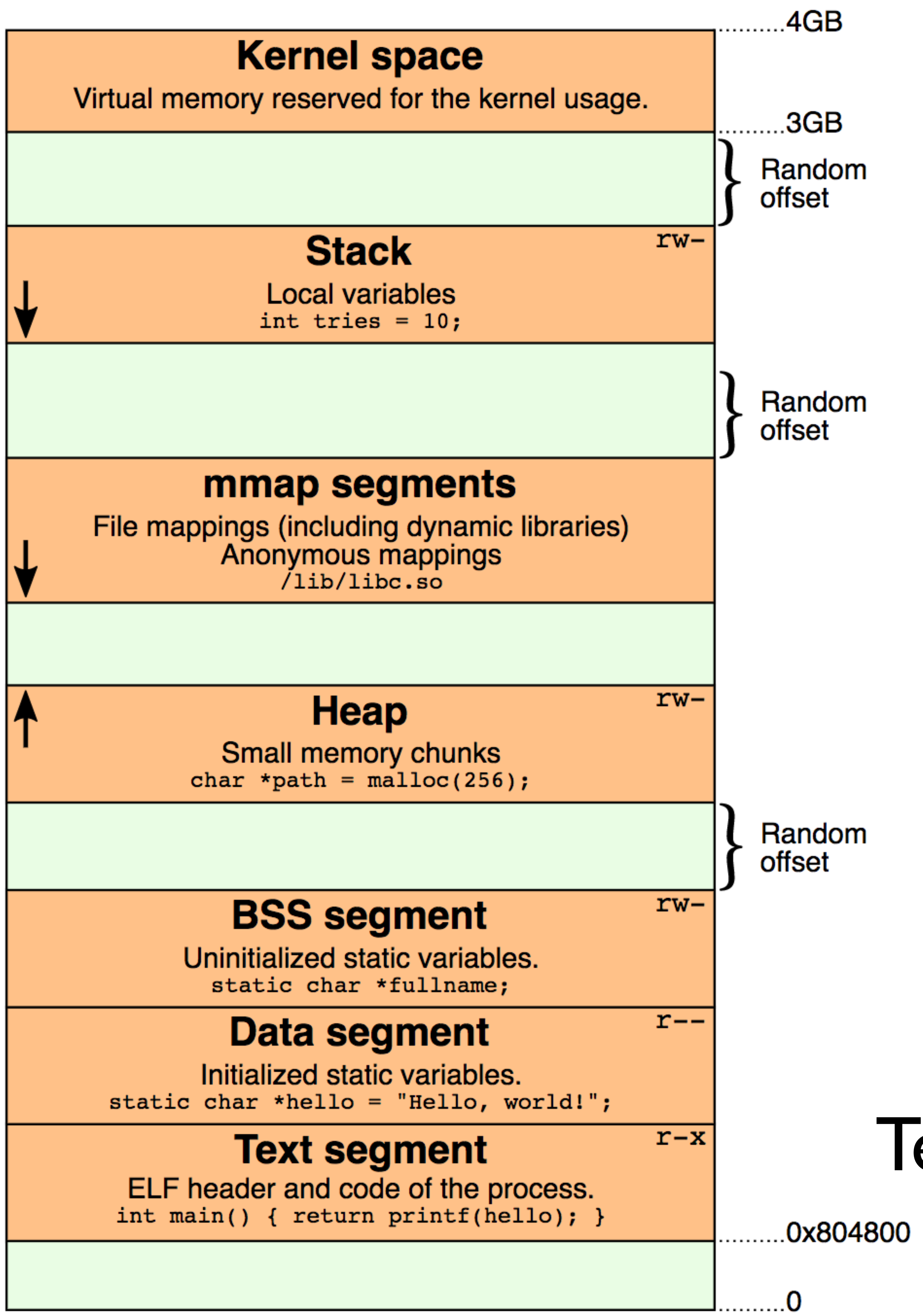
C: Malloc / free



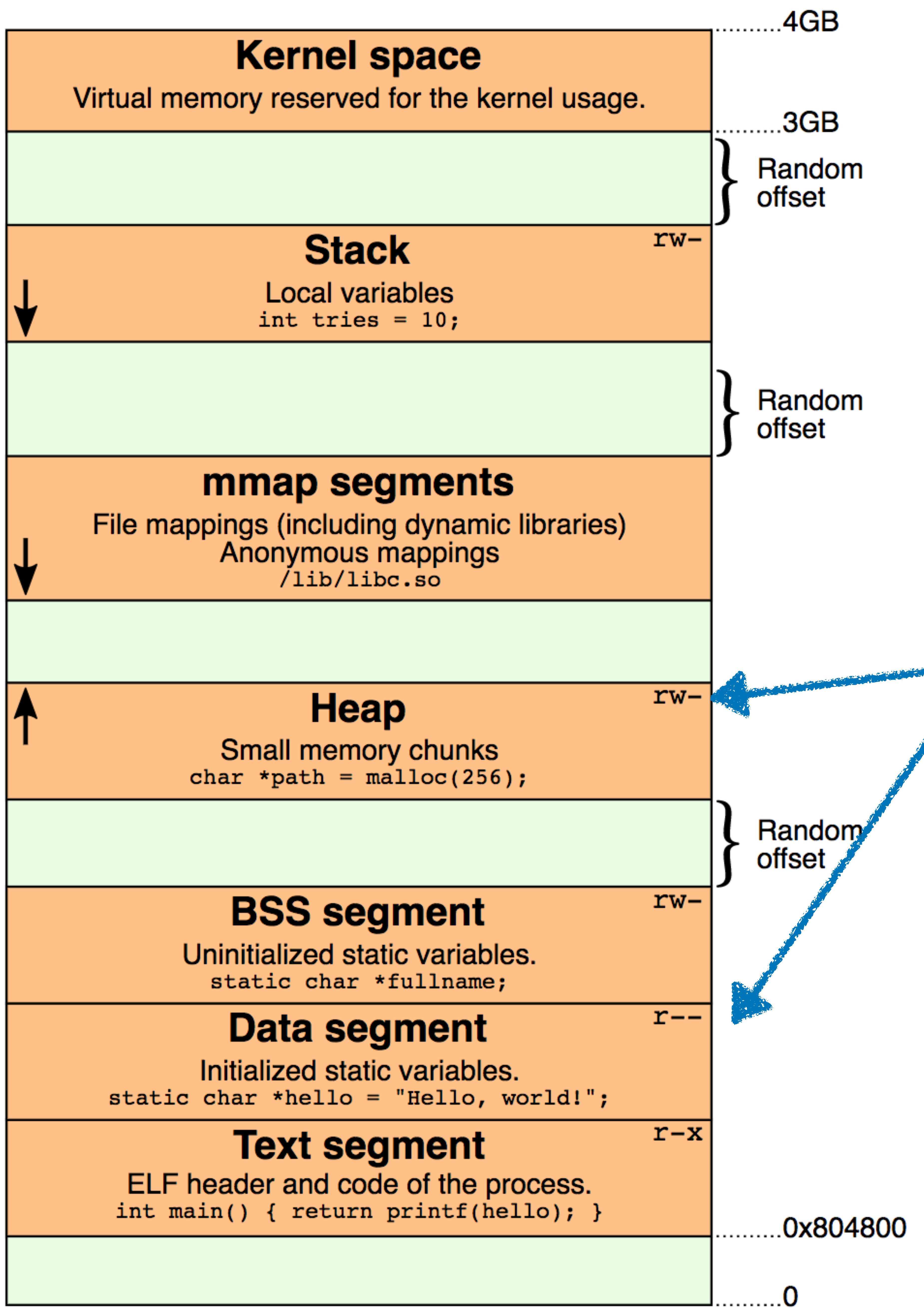
BSS: Uninitialized static vars (globals)



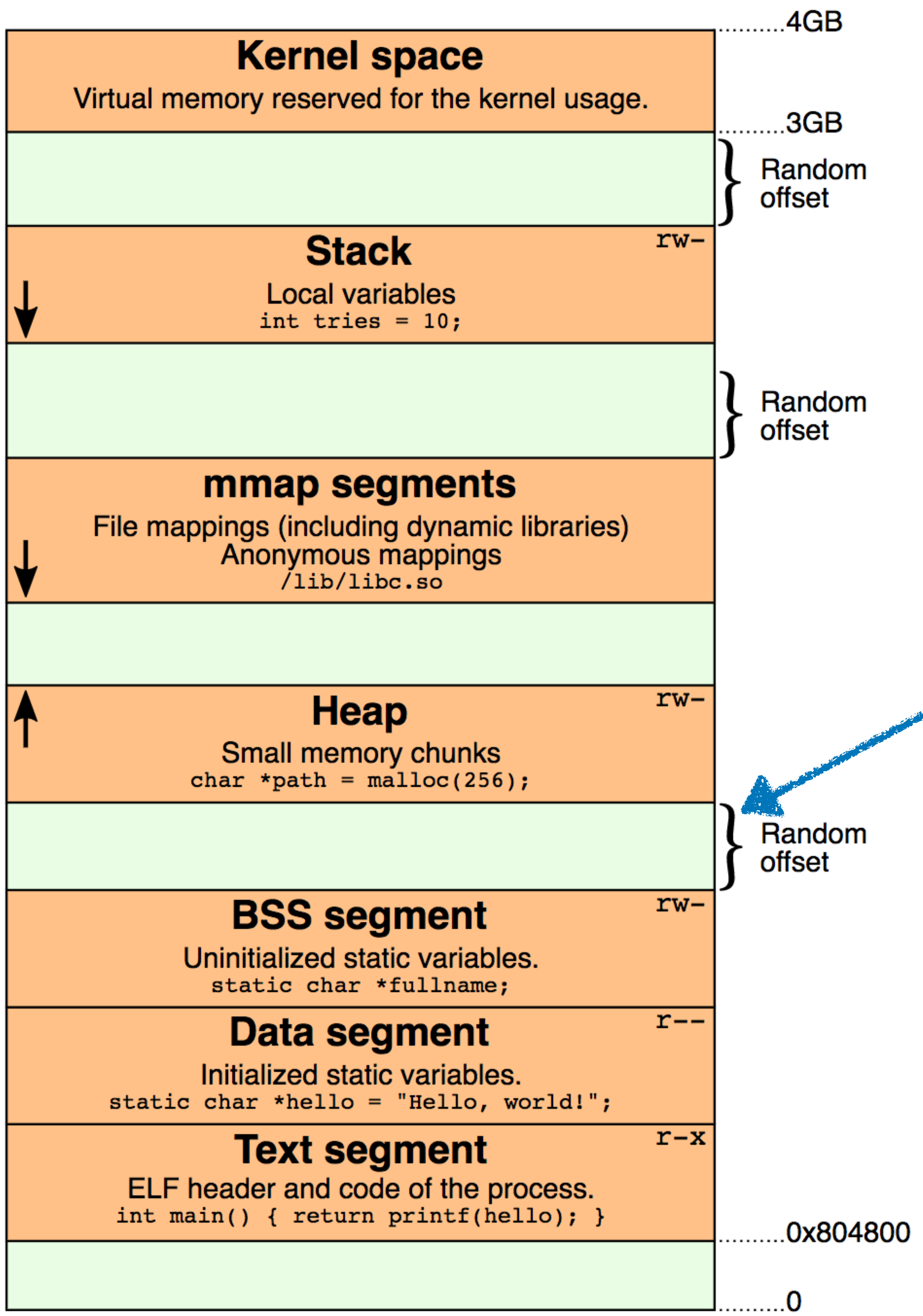
Data segment: initialized statics—e.g., constant strings



Text segment: program code



Note the **permissions**



This **random offset**
really security feature

Example: Summing an array

```
.text
.globl _main
_main:
  pushq %rbp
  leaq data(%rip), %rax    # rax -- Pointer to count
  movq $5, %rbx            # rbx -- Size of count array
  movq $0, %rcx            # rcx -- Index var for loop
  movq $0, %rdx            # rdx -- Sum total of array
_loop:
  cmp %rcx, %rbx
  je _end_of_loop
  mov (%rax, %rcx, 8), %r8 # Loads *(rax + %rcx * 8) -> %r8
  addq %r8, %rdx
  addq $1, %rcx
  jmp _loop
_end_of_loop:
  movq %rdx, %rdi
  call _exit

.data
data:
  .quad   4, 2, -3, 1, 8  # Declares an array of 8-byte values
```

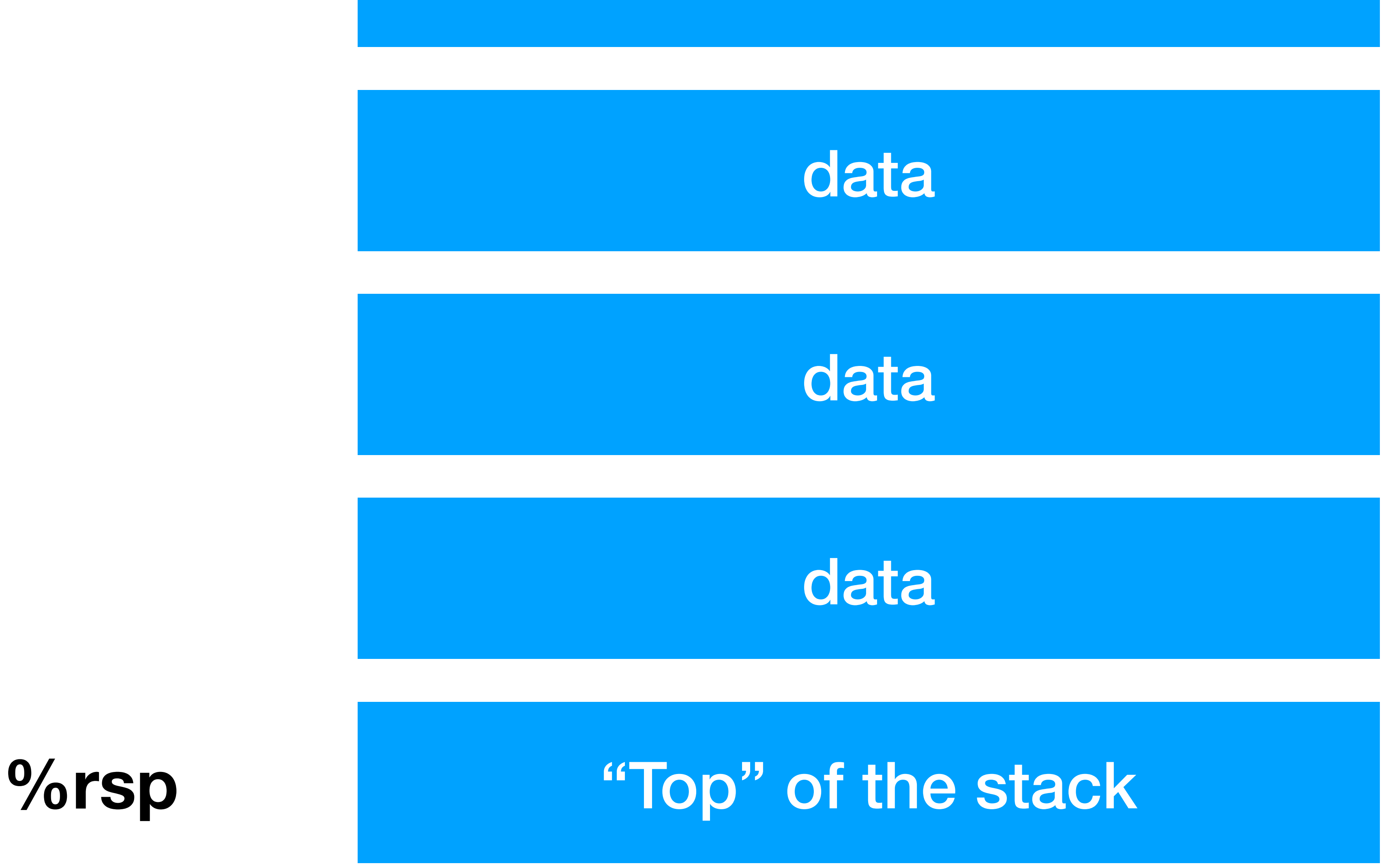
What do you do when you run out of registers..?

(There are only a limited number, so you **will** run out!)

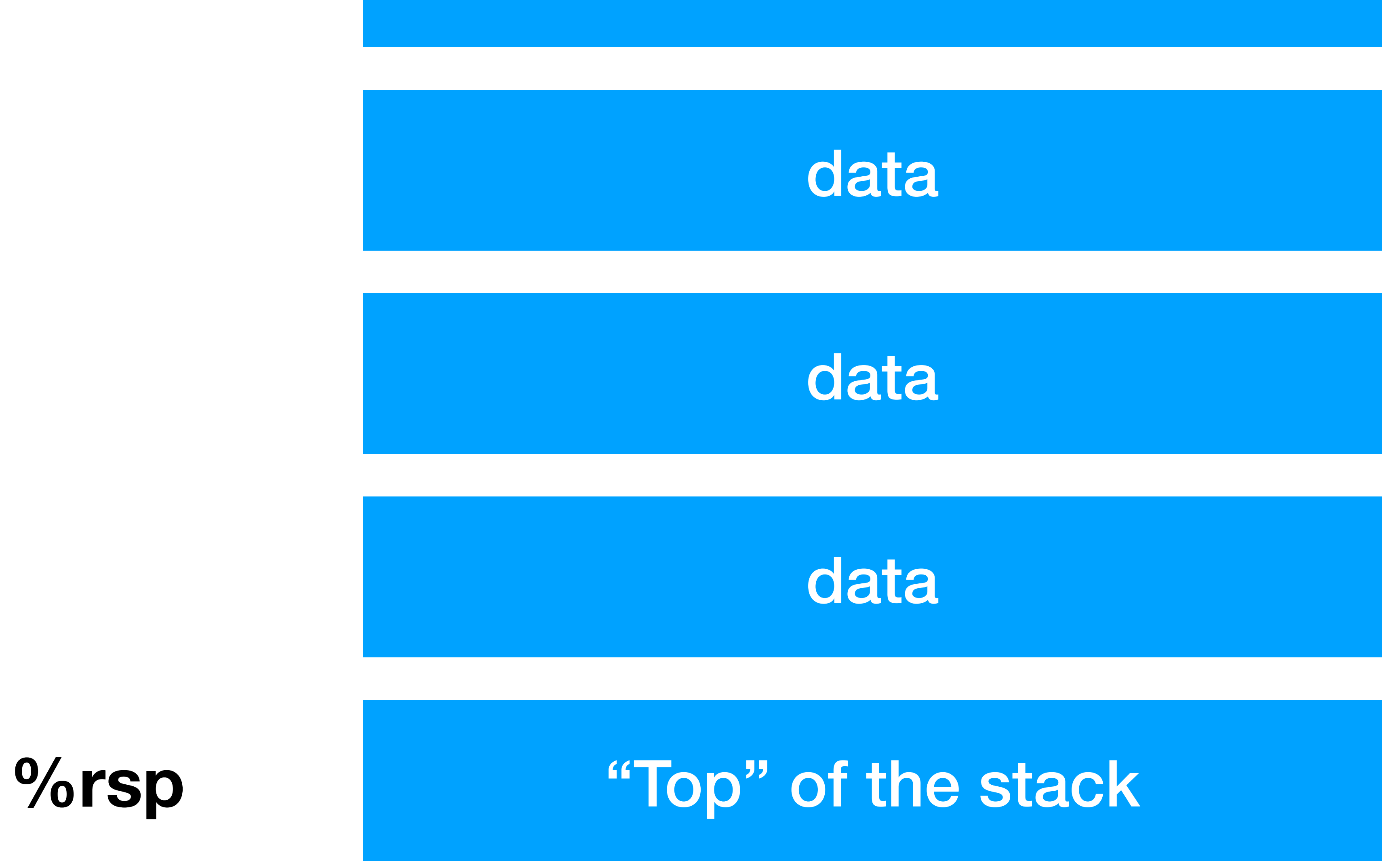
What do you do when you run out of registers..?

(There are only a limited number, so you **will** run out!)

Observation: can also use the **stack** to store data!

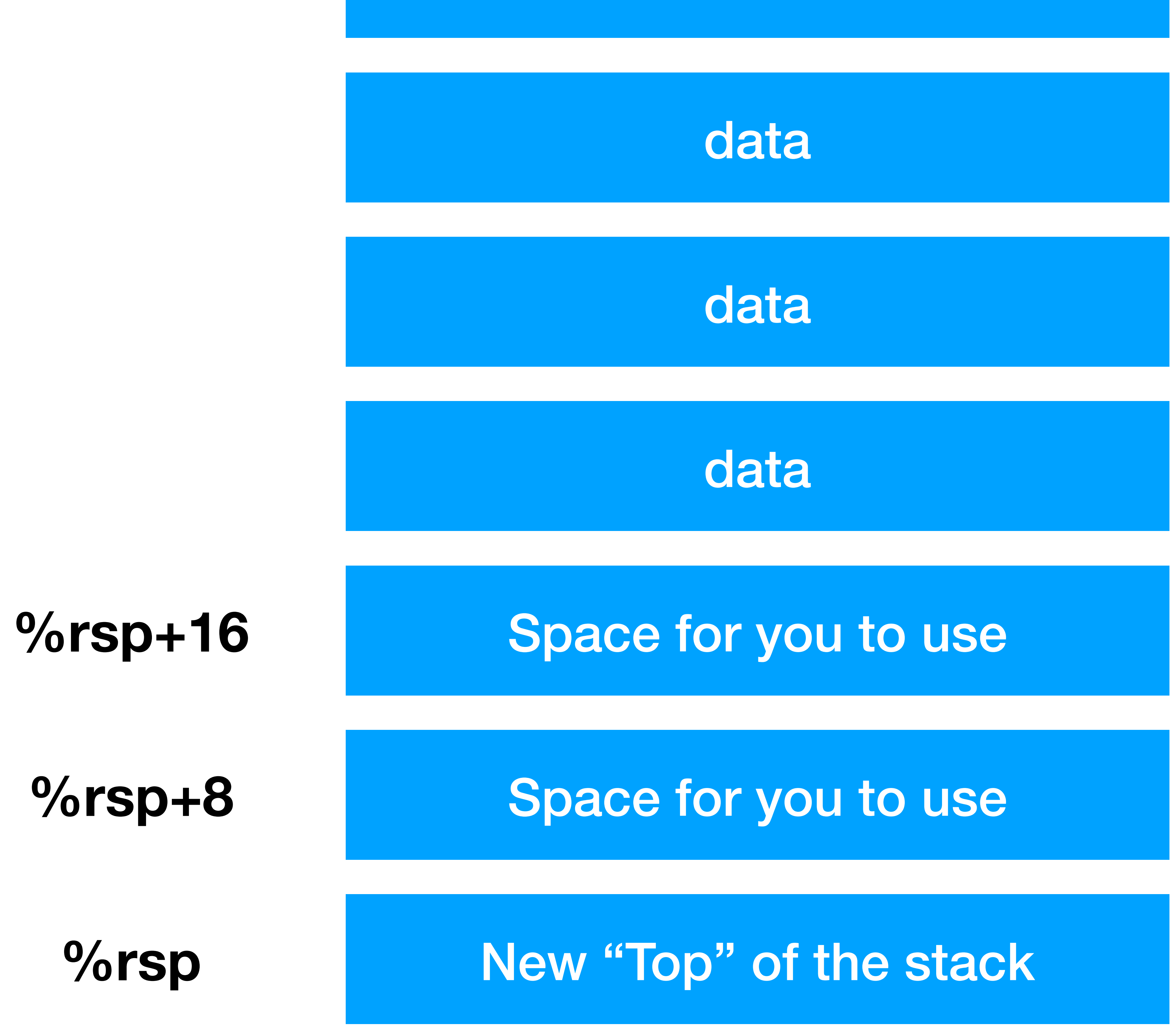


The **stack pointer** %rsp points at the top of the stack



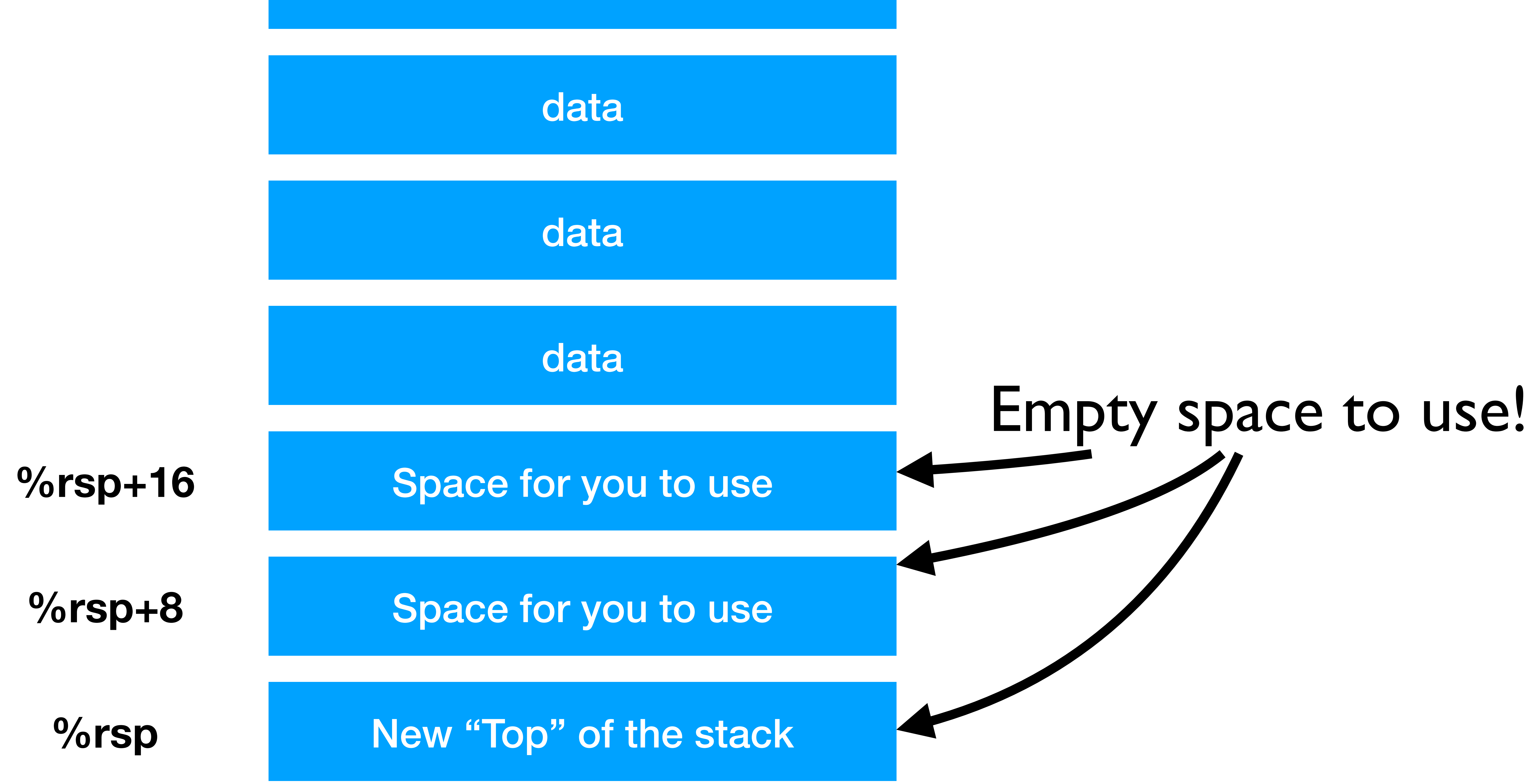
The **stack pointer** %rsp points at the top of the stack

If you want to store data on the stack, just subtract from %rsp and store there!



The **stack pointer** %rsp points at the top of the stack

If you want to store data on the stack, just subtract from %rsp and store there!

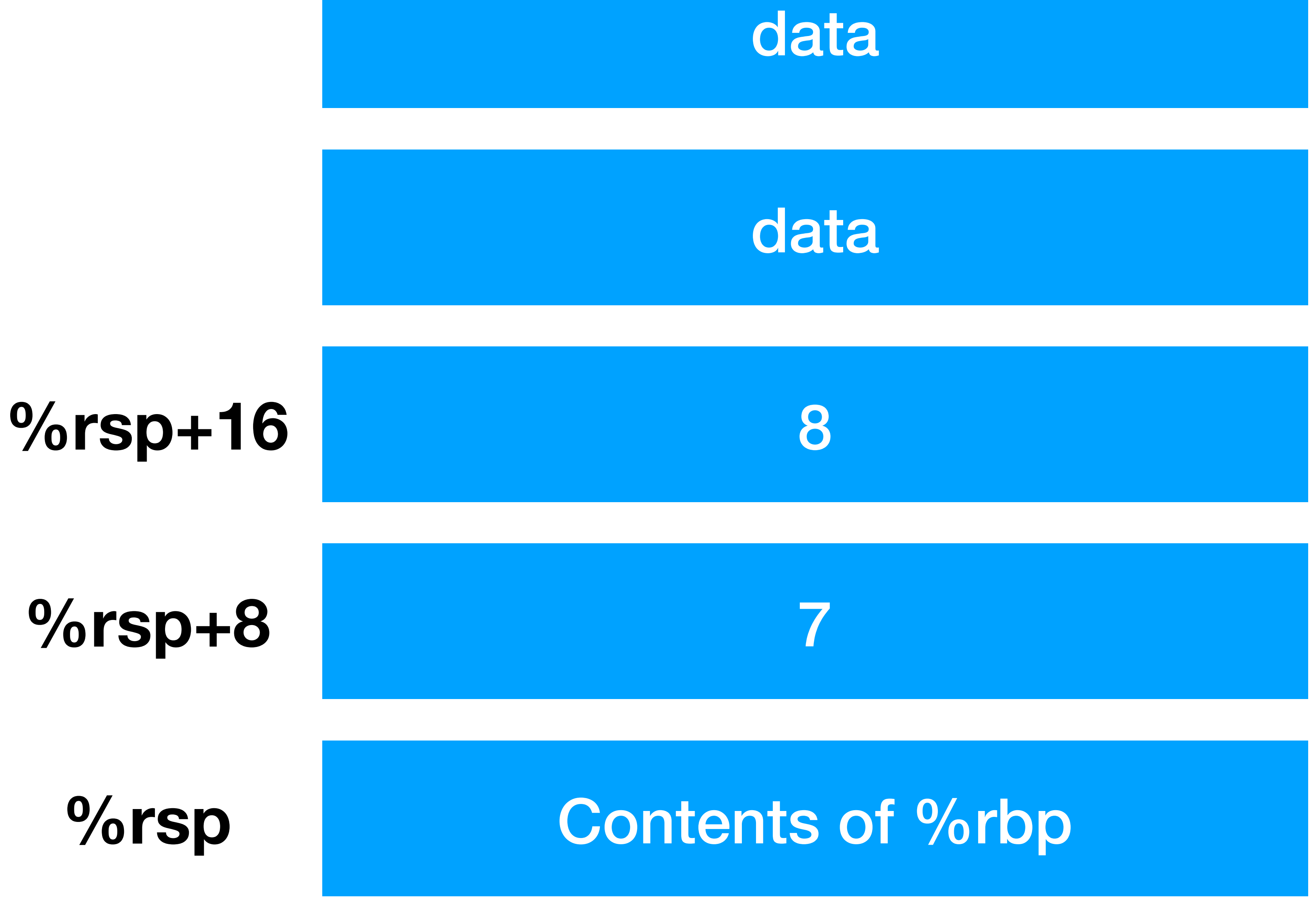


The **stack pointer** %rsp points at the top of the stack

If you want to store data on the stack, just subtract from %rsp and store there!

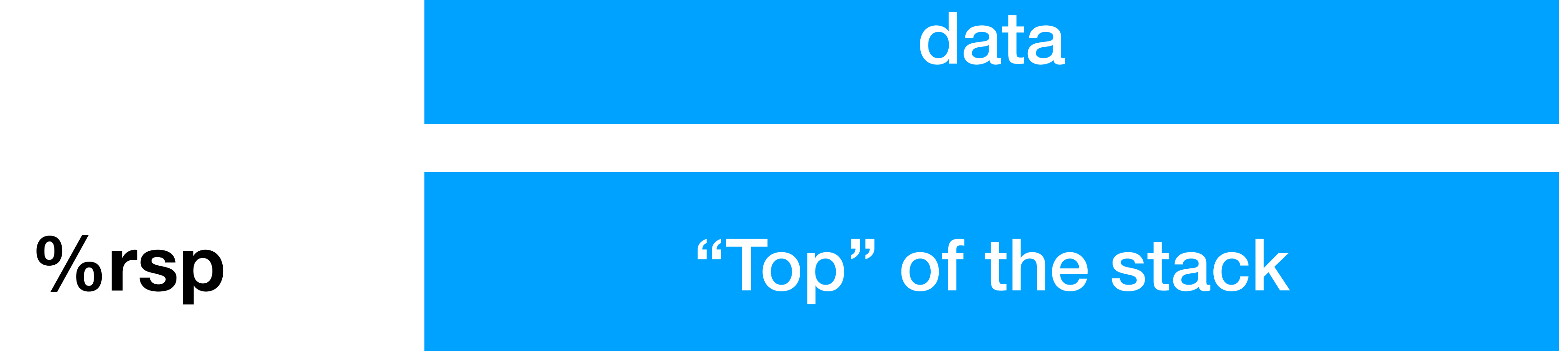
The "push" opcode decrements the stack and puts new data onto it

pushq %rbp

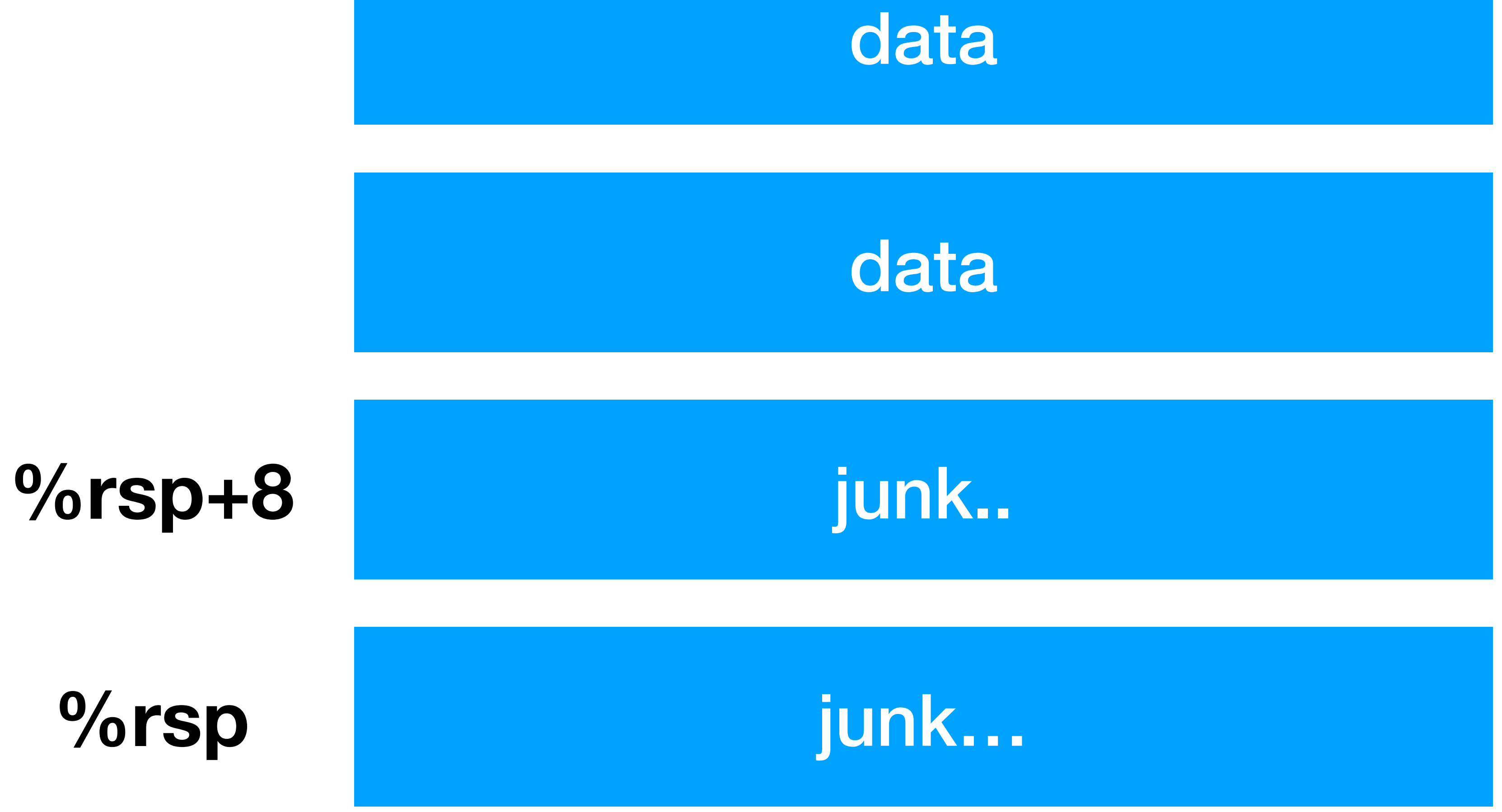


The "push" opcode decrements the stack and puts new data onto it

pushq %rbp




```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```




```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```


```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```


```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```

In many compilers (especially nonoptimizing ones), local variables are stored on the stack

(Even when they could be in registers!)

```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```

```
int main() {
  int x = 7;
  int y = 3;
  exit(x+y);
}
```

```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq %rsp, %rbp    # Move %rsp into the base pointer %rbp
  movq $7, (%rbp)    # Move 7 onto the top of the stack
  movq $3, 8(%rbp)   # Move 3 onto the next qword on the stack
  movq (%rbp), %rax  # Move *rsp into %rax
  movq 8(%rbp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```

Because the stack often grows up and down, programmers sometimes use %rbp

("**base pointer:**" points at **base** of local variables)

(Dereferences can use %rbp even when %rsp changes)

Because functions often store their local variables on the stack, a common “recipe” for writing a function is:

- Push %rbp onto the stack (save the caller’s %rbp)
- Subtract **x** bytes from the stack
  - Where x is the number of bytes taken by local variables
  - Often padded to the nearest 16-byte value for alignment
- Move %rsp into %rbp
- Each local variable is now at (%rbp), 8(%rbp), …

```
int foo() {
  int x;          // 4 bytes
  int y;          // 4 bytes
  char foo[16];   // 16 bytes
  double z;       // 8 bytes
}
```

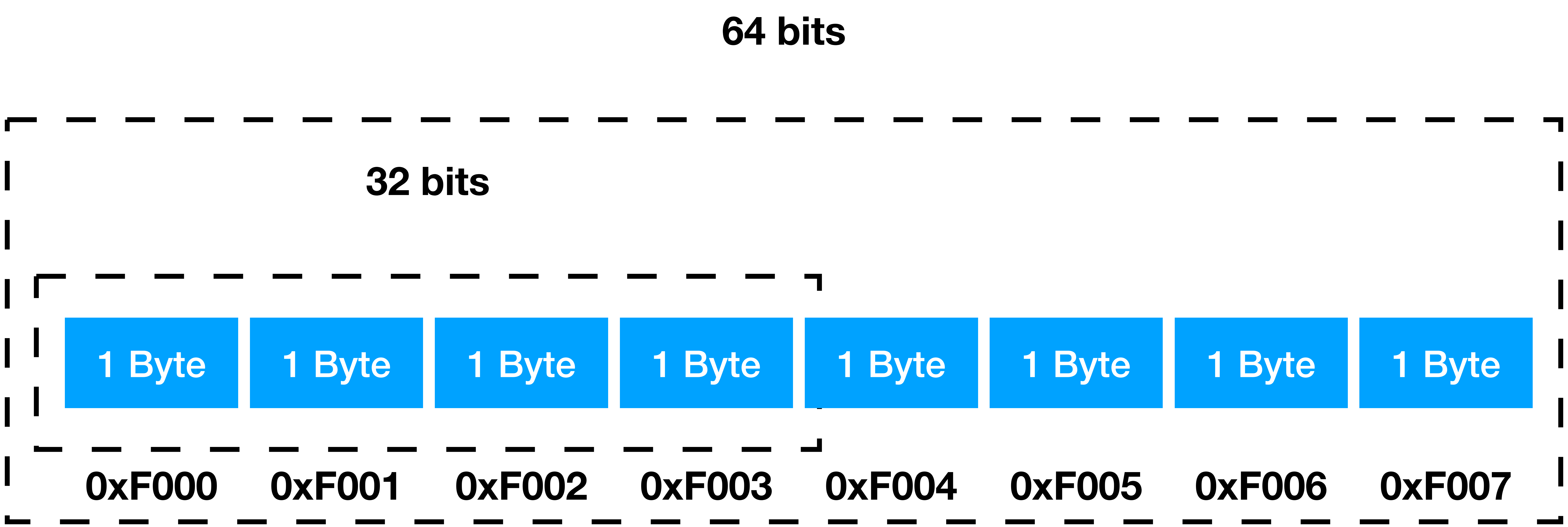
```
_foo:
  pushq %rbp
  subq  $16, %rsp
  movq  %rsp, %rbp
  # z is (%rbp)
  # foo is 8(%rbp)
  # y is 24(%rbp)
  # z is 28(%rbp)
```

Important aside: Alignment

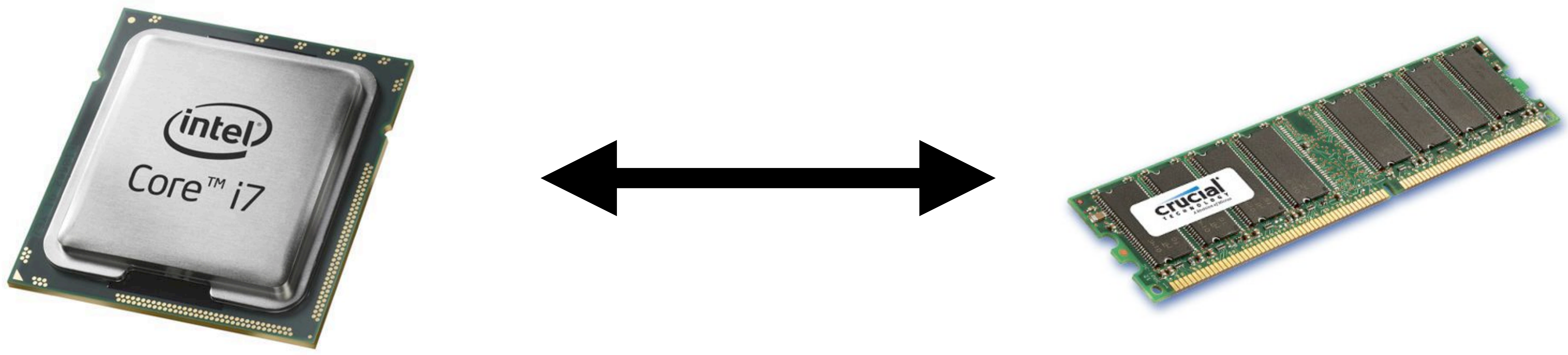
Concept of laying out data in memory to respect constraints of the ISA's memory access conventions

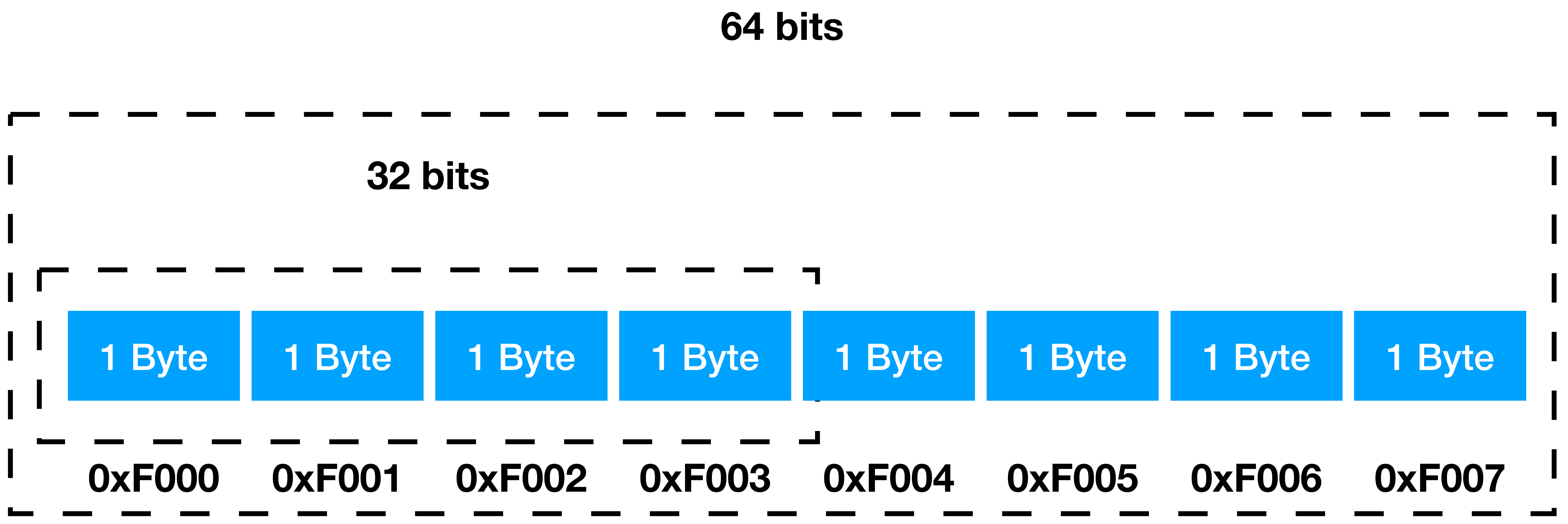
Typically, an n-byte datatype will be aligned on an n-byte boundary (where n is 1,2,4,8,16,…)

E.g., a **double** in C++ is 8-bytes in size, meaning it must sit at a memory address which is divisible by 8 (0x00, 0x08, 0x10, …)



Your processor talks to RAM via a bus

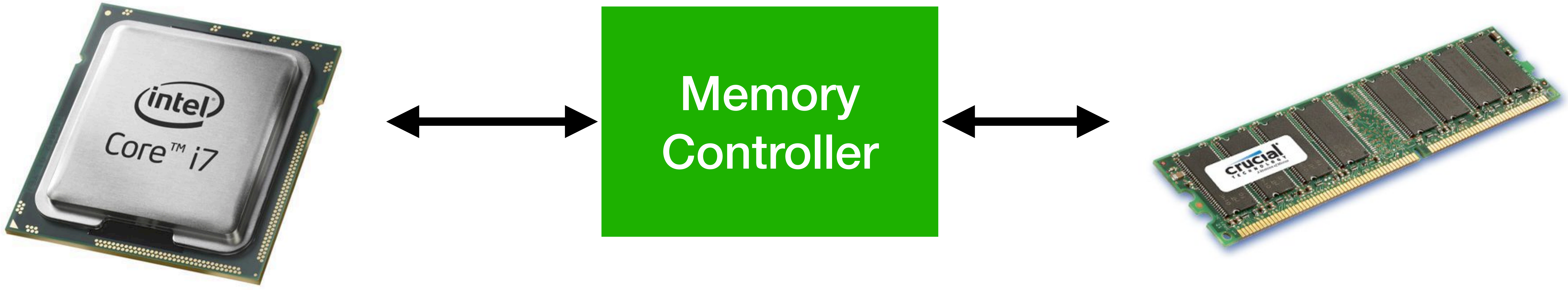


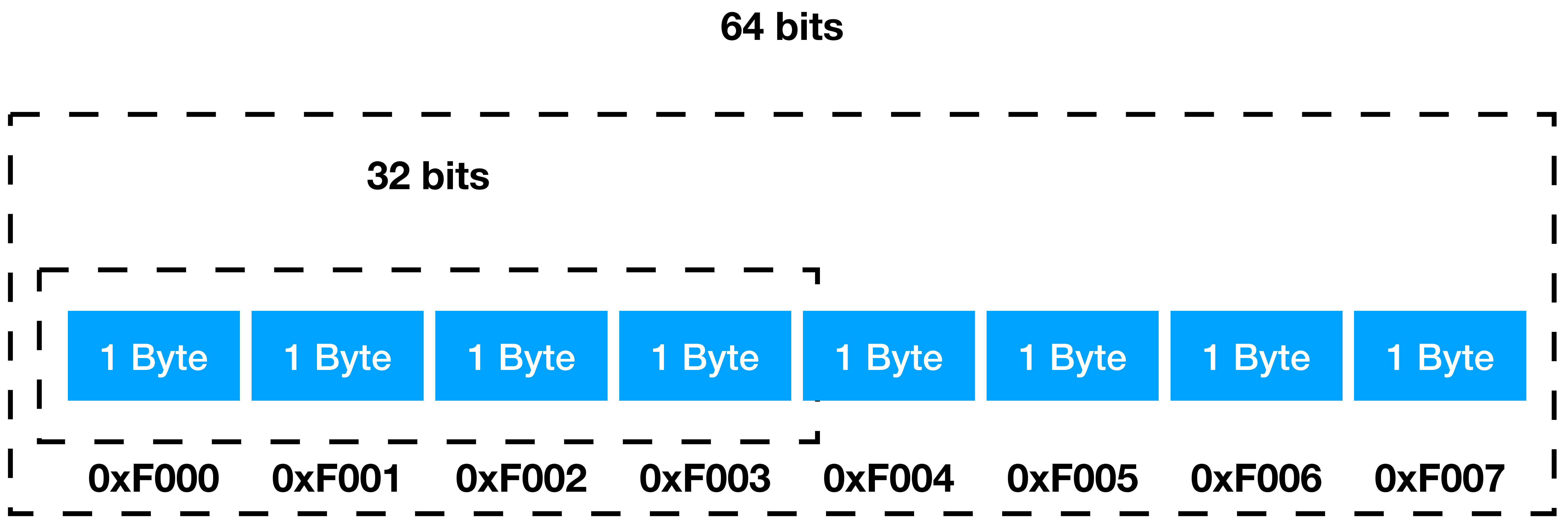


Your processor talks to RAM via a bus

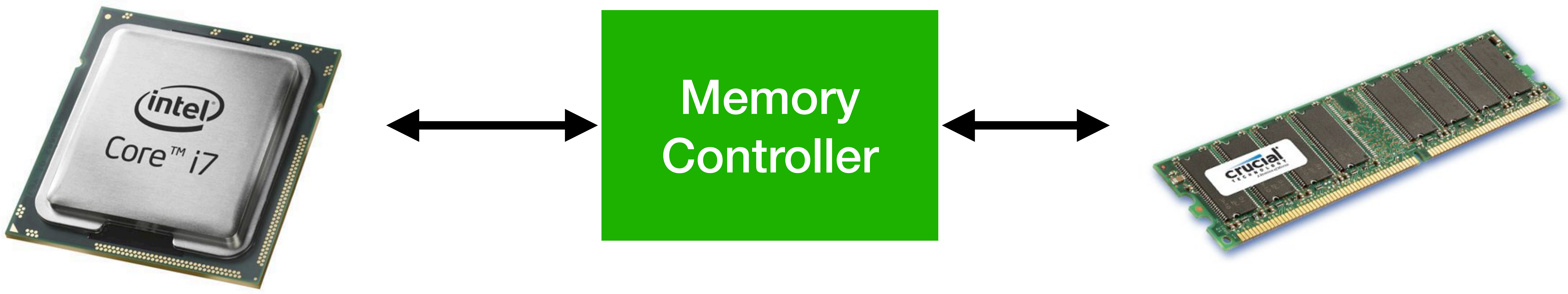
The memory controller interfaces the RAM banks to the CPU

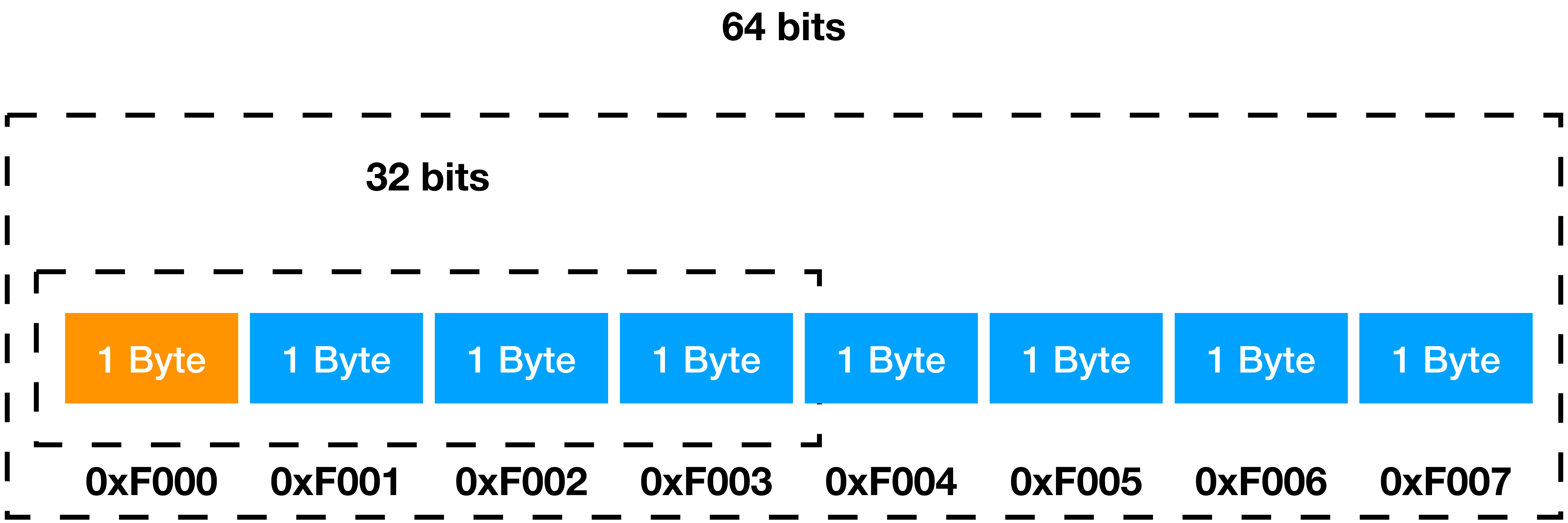
(E.g., what if multiple CPUs access same RAM at once)



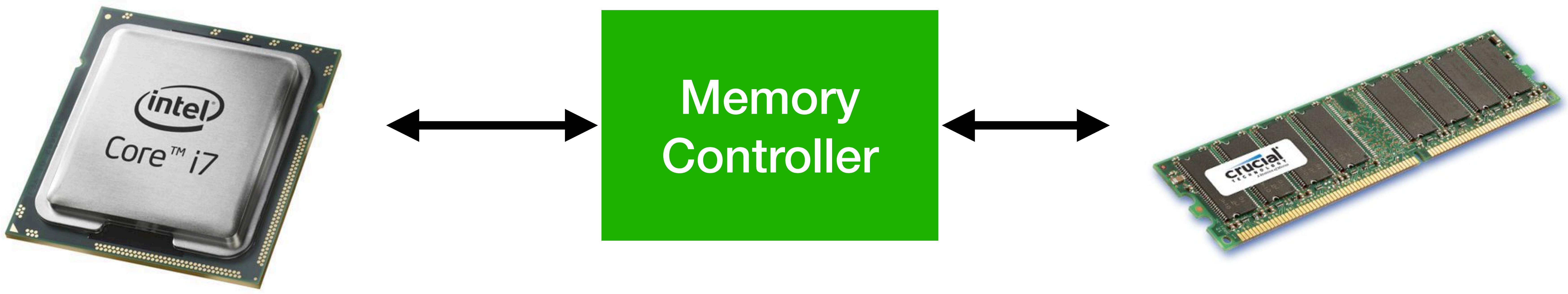


It makes the memory controller circuitry simpler when it only allows accessing memory at an address which is a multiple of 8 (etc..)





Valid start of an 8-byte datatype

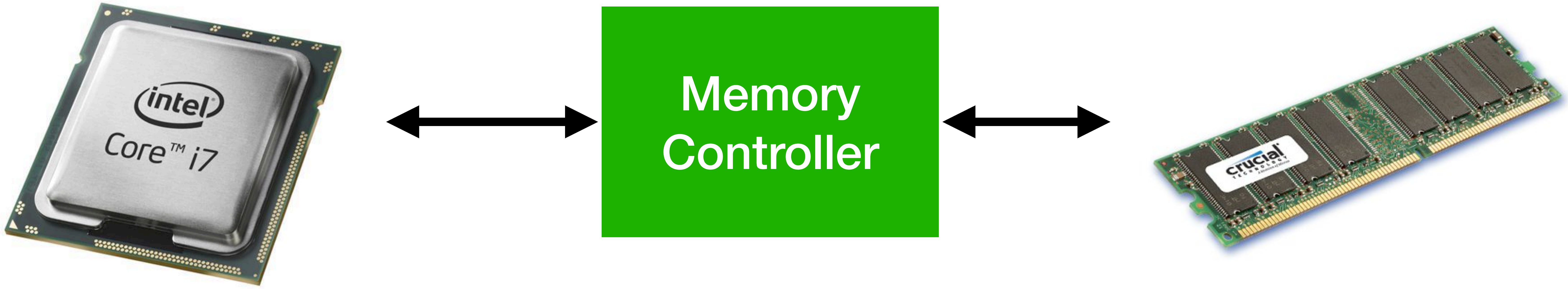


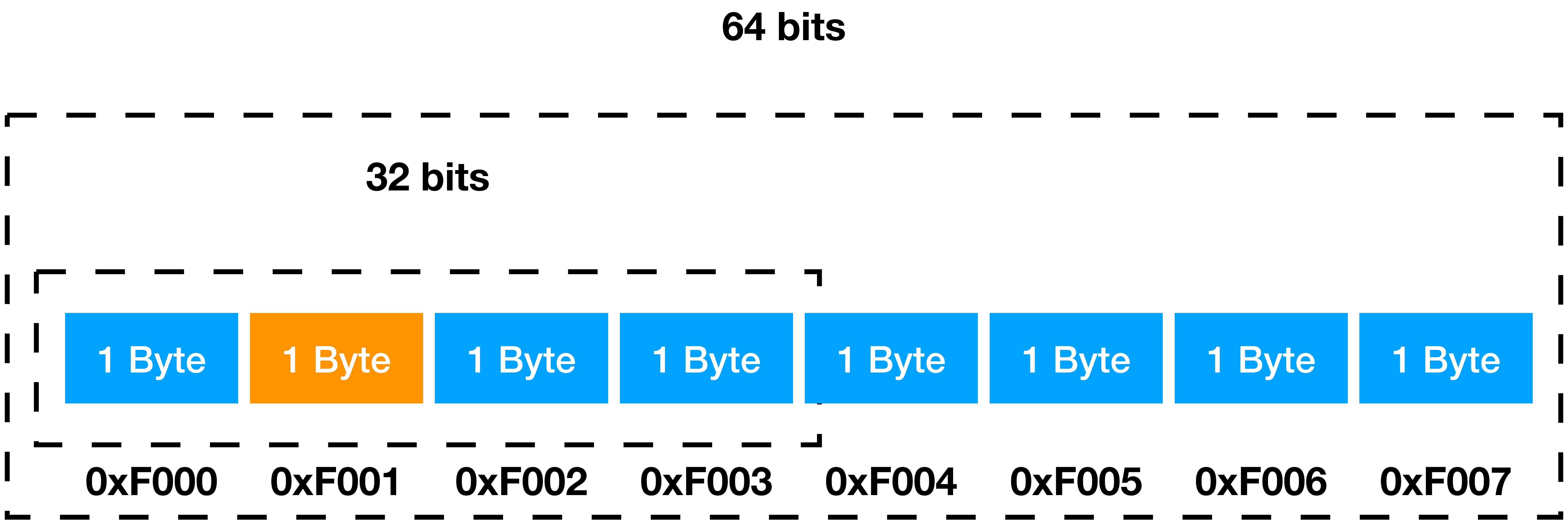
64 bits

32 bits

1 Byte	1 Byte	1 Byte	1 Byte	1 Byte	1 Byte	1 Byte	1 Byte
0xF000	0xF001	0xF002	0xF003	0xF004	0xF005	0xF006	0xF007

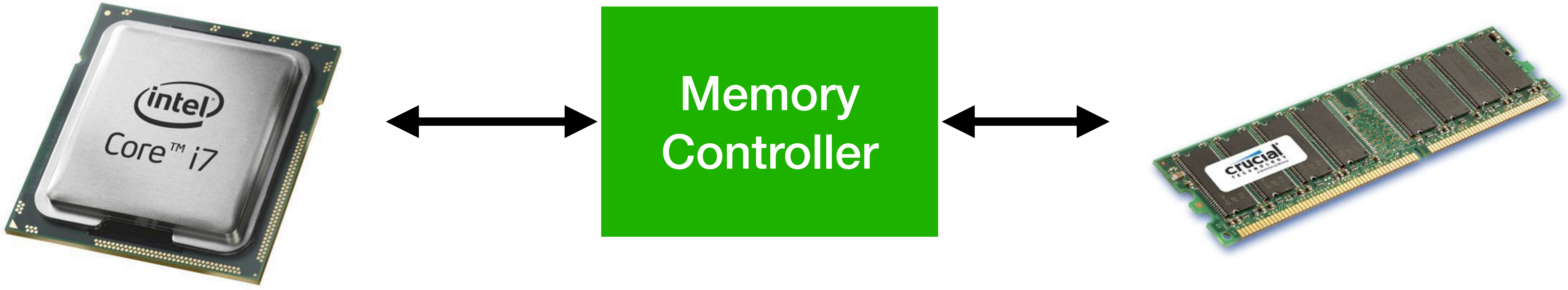
Invalid: address (0xF001) is not multiple of 8





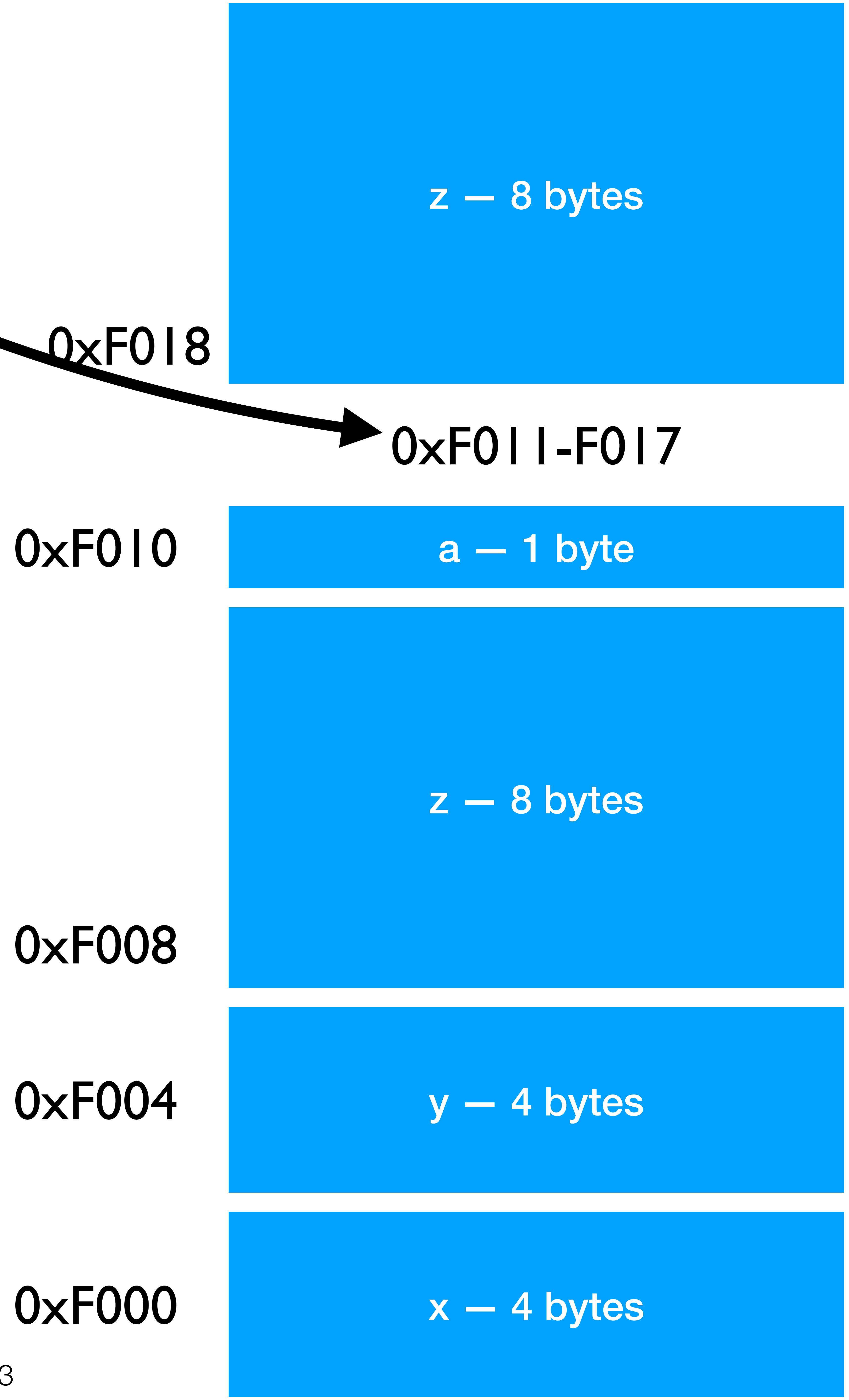
Invalid: address (0xF001) is not multiple of 8

(In this case, the processor actually does **two** fetches.
One from 0xF000 to get 0xF001-0xF007, One to get 0xF008-…)



Empty space for alignment

```
struct Foo {
  int x;
  int y;
  char *z;
  char a;
  int *num;
}
```



Structs laid out sequentially in memory, but alignment must be maintained!

Quiz: Which takes less space?

```
struct Foo {
  char y;
  int  z;
  int *num;
}
```

```
struct Foo {
  int *num;
  int  z;
  char y;
}
```

Quiz: Which takes less space?

```
struct Foo {
  char y; // 1 byte
  // 3 bytes empty space
  // int must be at addr
  // divisible by 4
  int  z;
  // 4 bytes empty space
  // after z, int* must
  // be on 8-byte boundary
  int *num;
}
```

```
struct Foo {
  int *num;
  // No empty space
  int  z;
  // No empty space
  char y;
}
```

Answer: right takes less, since plays better with alignment!

<END>
Crash Course on x86_64 assembly!

- Assembly language is not complicated, but it is tedious and technical; you don't need much beyond the basics here
- There is always a question: target LLVM, MLIR, ARM, etc.?
  - We target x86_64; why?
  - Answer: *lots* of x86_64 desktop / server code out there, common, real, and ugly ISA; prevalent documentation and examples online for things like ABI, etc.

Check your Knowledge

A few questions after this lecture:

- Explain the difference between lexical analysis and parsing
- What is the difference between source, object, and executable code?
- What is an ABI?
- Is there a difference between the ABI and the file format (ELF, Mach-O, Windows PE)?
- What is an addressing mode?
- Why do certain instructions support only a subset of addressing modes?

Ask me (kkmicins@syr.edu) if you want to discuss answers after class