Malware Analysis

CIS700 (Special Topics) — Fall 2021

Kris Micinski

Course Logistics

- Welcome to the course, I'm happy you're here!
- High-level: this is a **seminar course** (with some hands-on projects) introducing state-of-the-art techniques in the analysis of malware.
- Today: introduction to assembly
- Next few weeks: boot-up on C, assembly, debugging, traditional attacks (overflows, shell coding, etc..)
- Course website:
  - https://kmicinski.com/cis700-f21
- I will be making heavy use of Slack
  - **Please make sure you join the Slack right now!**

Course Grading and Notes

- We expect and **trust** that graduate students are expending significant effort in studying for the course in a way that will aid their own individual research efforts.
- Thus, grading for graduate courses is particularly non-adversarial in the sense that I would like to give everyone an A who demonstrates they significantly improved their research-level knowledge.
- Grading will be as follows:
  - 3 course projects (each worth 10%)
  - Paper write-ups and discussions (worth 50%)
  - One take-home final (worth 20%)

Course Delivery

- This course will be **part-lecture, part seminar**.
- In a lecture, instructor presents material and solicits participation. In seminar, students guide discussion informed by instructor's guidance.
    - Generally, Tuesdays will be lecture days and Thursdays will be paper discussion days.
- Slides will likely be **very terse** and I expect you will ask questions.
- I would like each lecture to have a **scribe**. Every must scribe **at least twice**. A scribe takes thoughtful notes on the lecture so we can post them later.

Course Grading and Notes

- We expect and **trust** that graduate students are expending significant effort in studying for the course in a way that will aid their own individual research efforts.
- Thus, grading for graduate courses is particularly non-adversarial in the sense that I would like to give everyone an A who demonstrates they significantly improved their research-level knowledge.
- Grading will be as follows:
  - 3 course projects (each worth 10%)
  - Paper write-ups and discussions (worth 50%)
  - One take-home final (worth 20%)

Course Projects

- Three projects. I am not quite sure what these will be yet. Topics may include:
  - Stack overflow exploitation / shell coding and stack overflow prevention
  - SQL injection or other more modern web attacks
  - Reproducing an attack from a paper
  - (Manual/automated) ROP synthesis and exploitation

Topics (**Very** tentative)

- Week 1 — C / assembly background
- Week 2 — Spatial and temporal safety
- Week 3 — Shellcoding, ASLR, probabilistic defenses
- Week 4 — Return-to-libc and ROP
- Week 5 — Modern ROP synthesis and exploit generation
- Week 6 — Symbolic execution and SMT
- Week 7 — Modern binary symbolic execution techniques (angr, BAP, etc..)
- Week 8 — Datalog, Datalog Disassembly, and Horn-SAT-based binary analysis
- Week 9 — Decompilation and sound decompilation.
- Week 10 — Machine learning for malware classification.
- Week 11 — Neural inference of binaries for decompilation, identifier reversing, etc…
- Week 12 — Usable reverse engineering tools
- Week 13 — Scriptable, declarative, and compilable binary analyses.
- Week 14 — Project presentations

Memory-Based Attacks

Assembly Review

By which I mean x86-64 assembly...

Note: you won't have to write significant amounts of assembly for this course, but you will need to be able to read small pieces of it and figure out what it's doing...

Note: you won't have to write significant amounts of assembly for this course, but you will need to be able to read small pieces of it and figure out what it's doing...

Also note: I will be discussing x86 assembly, although it's arguably a dying language (behold—ARM!). x86 assembly is still the bulk of what a reverse engineer would see, so I think it makes sense to teach that...

Registers

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)

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)

IP: instruction pointer

Points at current instruction, incremented after each instruction

FLAGS: holds flags

Set on subtraction, comparison, etc..

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

Also other registers: bp, sp, di, si

As time progressed, also added 32-bit registers: eax, ebx, ecx, edx

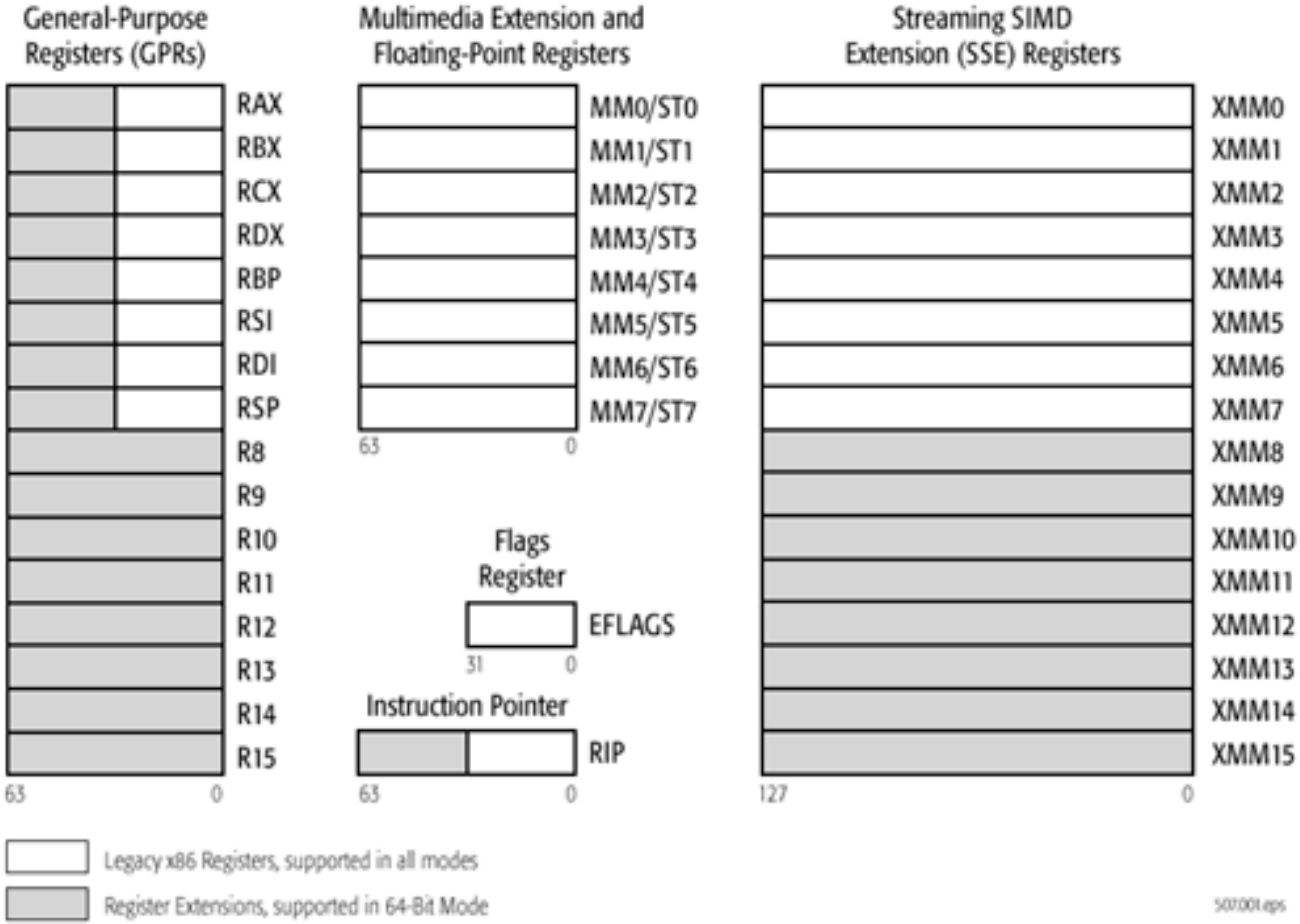
In past few years, 64-bit registers: rax, rbx, rcx, rdx

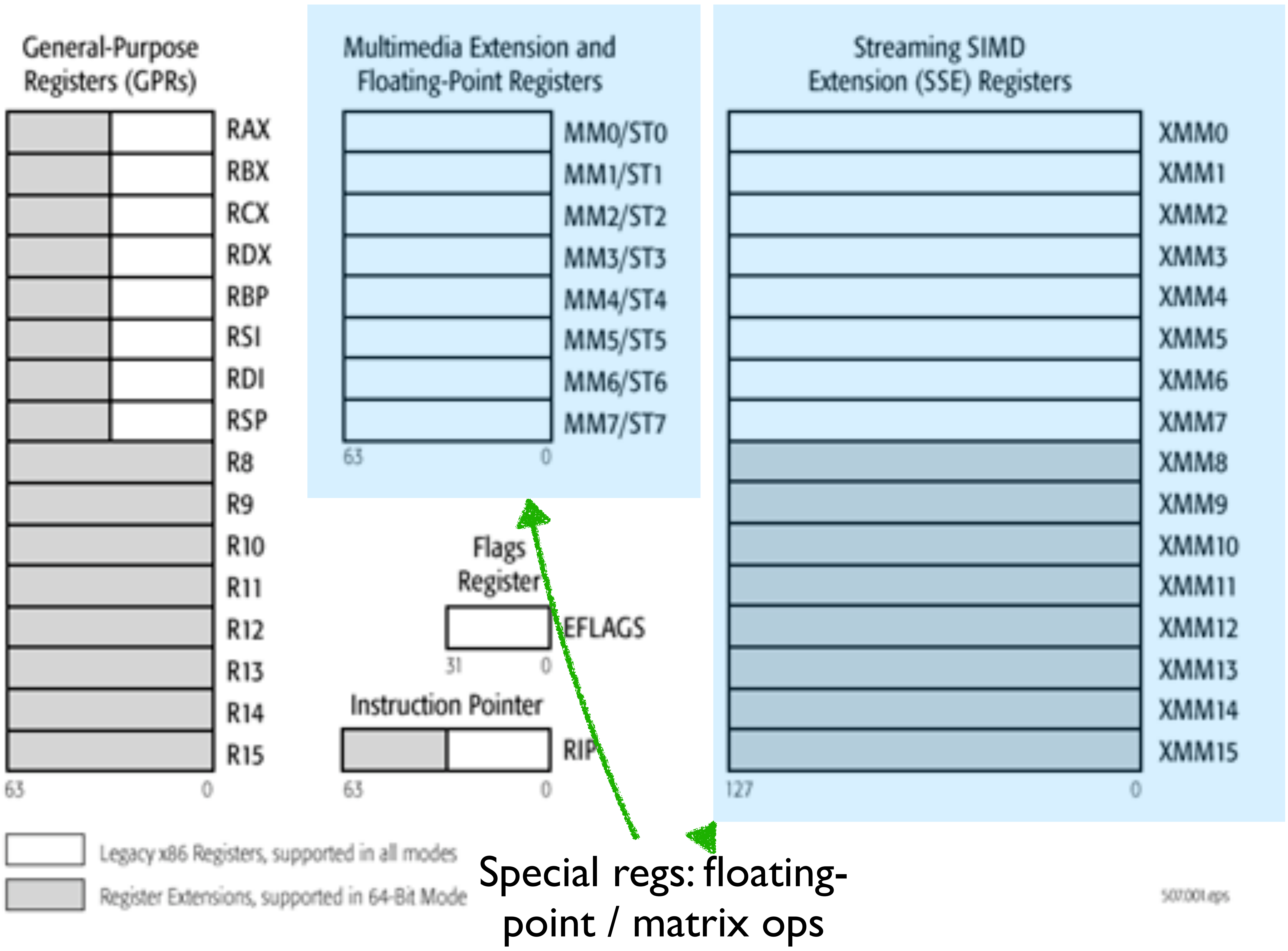
(Also 64-bit versions: rip, etc..)

We'll pretty much exclusively use 64-bit registers!

Note RAX is an **extension** of EAX

If you change EAX, you change lower 32 bits of RAX


General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
Multimedia Extension and Floating-Point Registers
MM0/ST0
MM1/ST1
MM2/ST2
MM3/ST3
MM4/ST4
MM5/ST5
MM6/ST6
MM7/ST7
63
0
Flags Register
EFLAGS
31
0
Instruction Pointer
RIP
63
0
Streaming SIMD Extension (SSE) Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 Registers, supported in all modes
Register Extensions, supported in 64-Bit Mode



General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
Multimedia Extension and Floating-Point Registers
MM0/ST0
MM1/ST1
MM2/ST2
MM3/ST3
MM4/ST4
MM5/ST5
MM6/ST6
MM7/ST7
63
0
Flags Register
EFLAGS
31
0
Instruction Pointer
RIP
63
0
Streaming SIMD Extension (SSE) Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 Registers, supported in all modes
Register Extensions, supported in 64-Bit Mode
Special regs: floating-point / matrix ops

To represent `0x1234567890abcdef`

12	34	56	78	90	ab	cd	ef

Most Significant Byte — Least Significant Byte 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**

`0x1234567890abcdef`

Exercise with partner

Instructions

Binary code is made up of giant sequences of "instructions"

Modern Intel / AMD chip has hundreds of them, some very complex

Moving memory around   Arithmetic   Branch / If

Matrix operations   Atomic-Instructions

Transactional memory instructions

Encoded as binary (as you may have seen from hardware-design course)

We (humans) write in a format named "assembly"

Confusingly: two types of assembly

AT&T	Intel
`mov 5, %rax`	`mov rax, 5`

I will basically always use AT&T

(Since that's what's used in GNU toolchain)

Several **addressing modes**

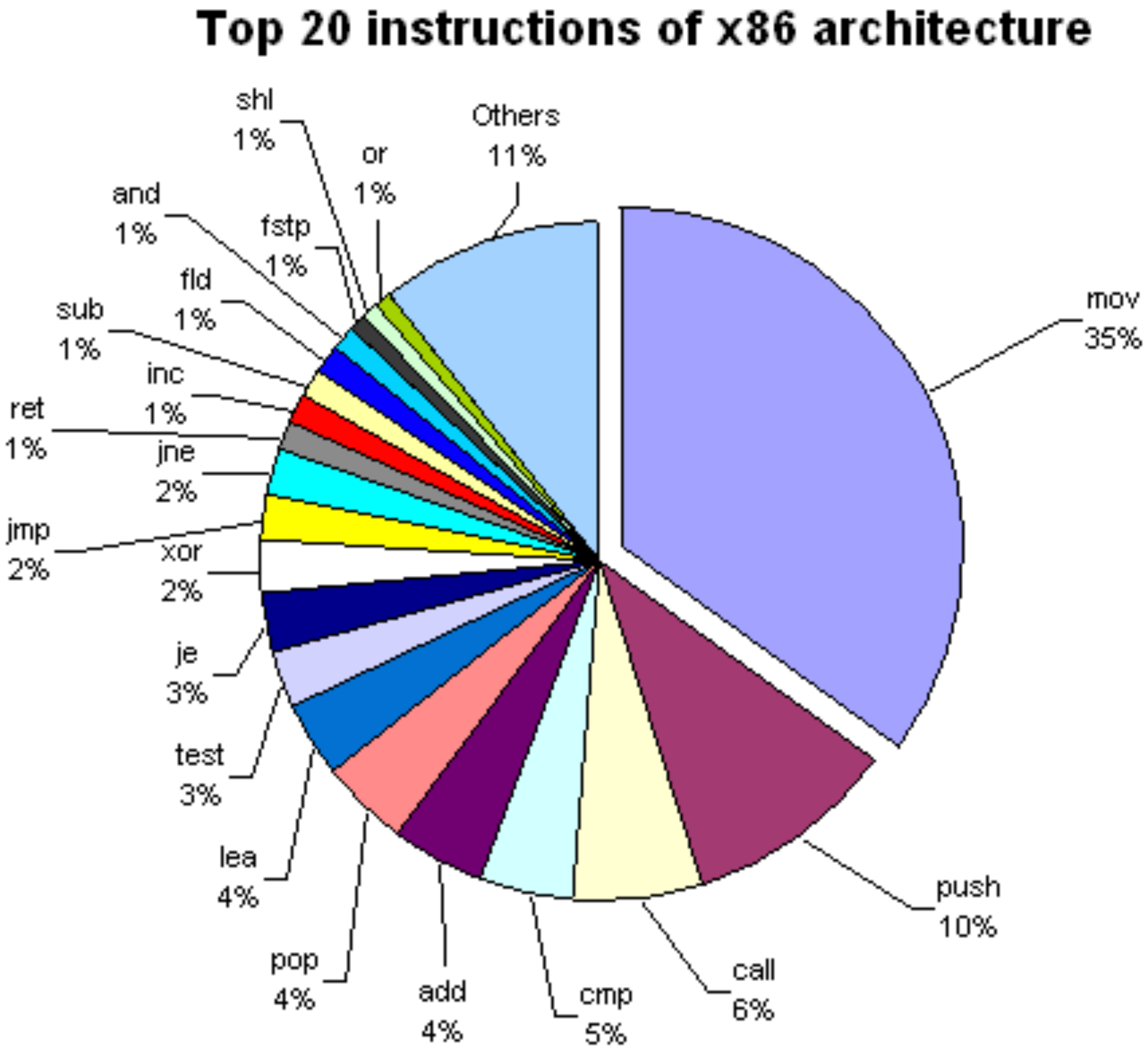
"Move the value from register rax into the register rbx"

Opcode name

Destination

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

Source



Plurality of instructions are **mov**s

Then **push**

Then **call**

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
```

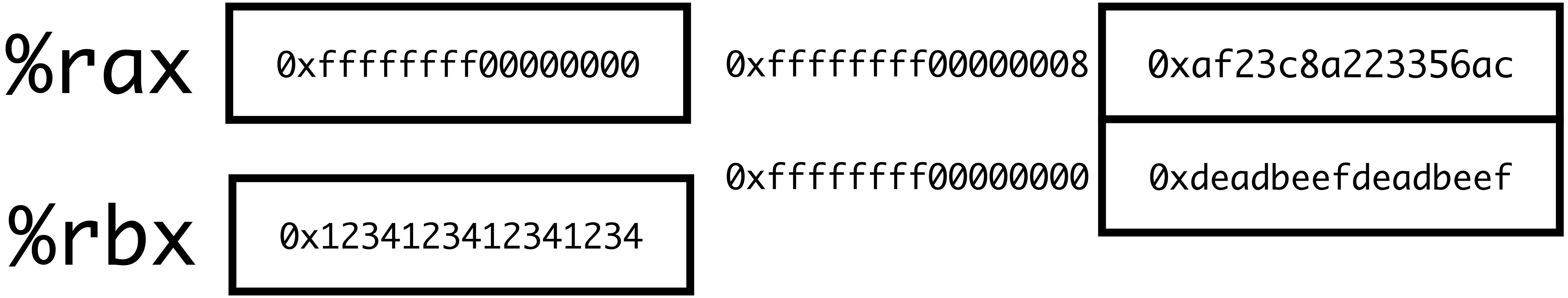
"Move the value **at address** %rax into register %rbx"

Opcode name

Destination

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

Source



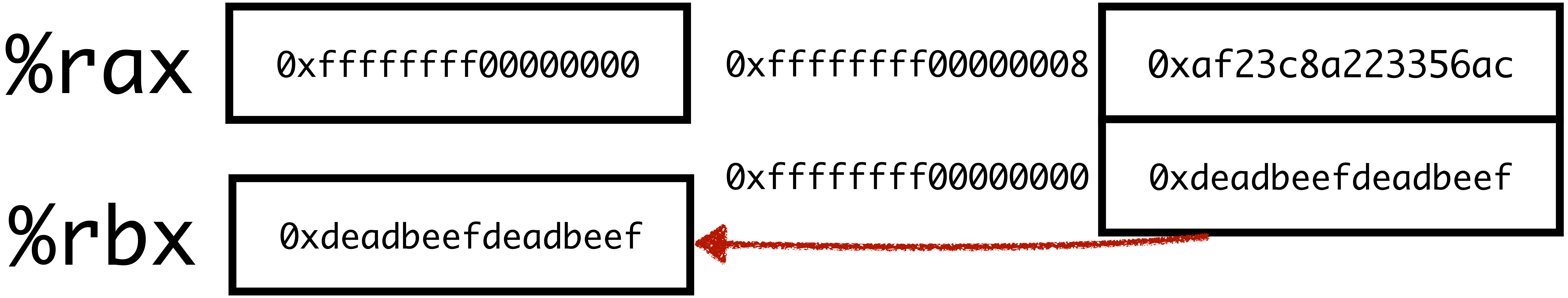
"Move the value **at address** %rax into register %rbx"

Opcode name

Destination

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

Source



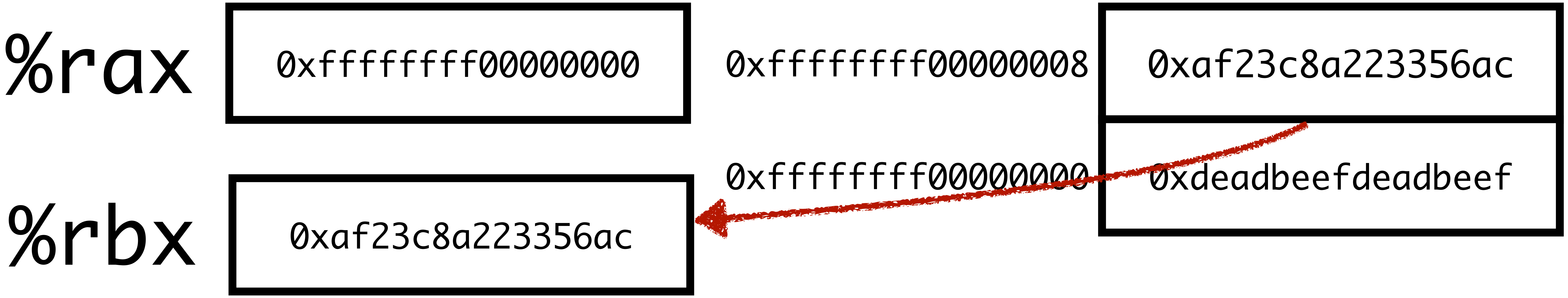
“Move the value **at address** %rax+8 into register %rbx”

Opcode name

Destination

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

Source



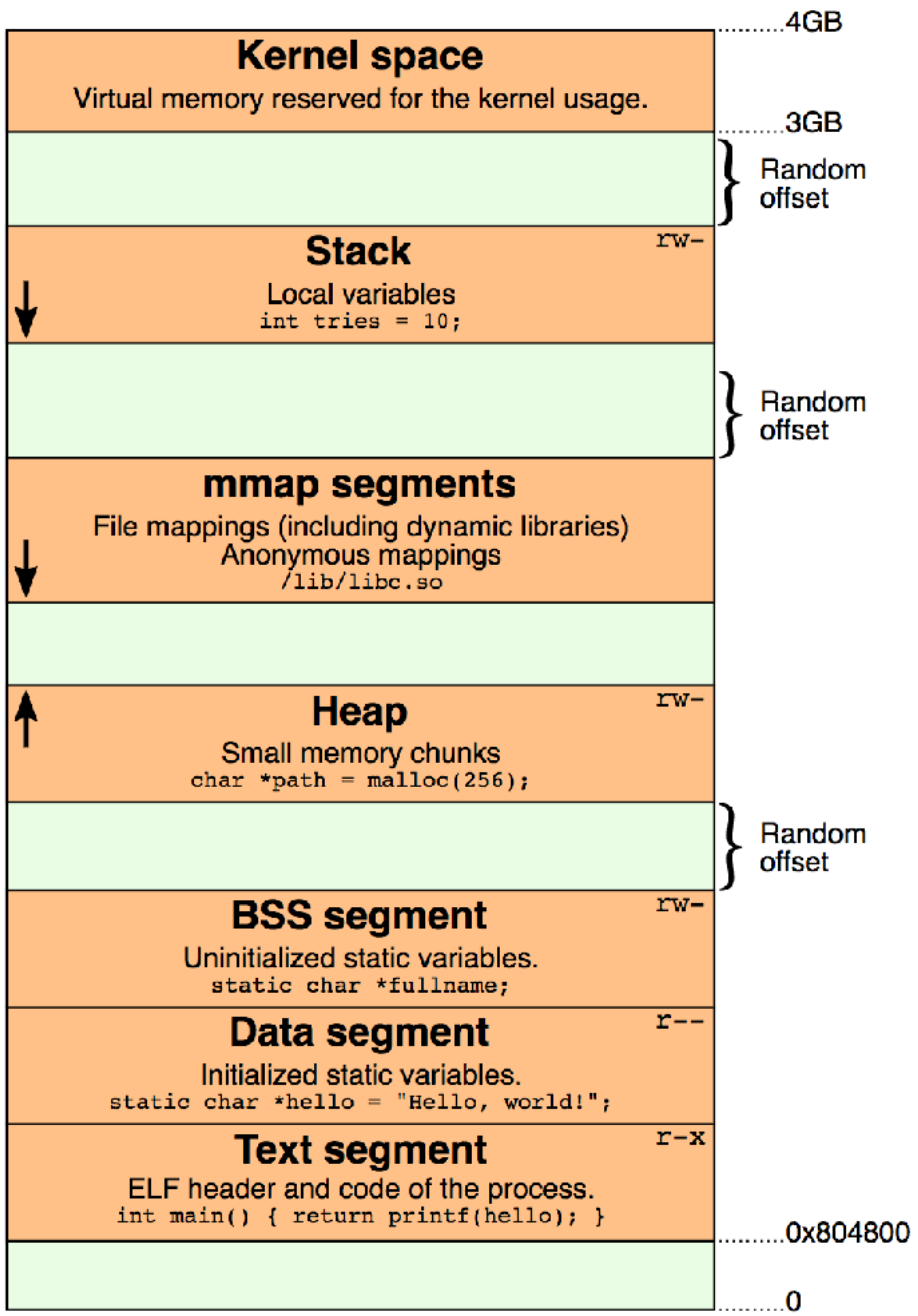
A few other more complicated ones that allow
you to add registers, offsets, etc…

Different instructions allow different addressing-modes

Memory is divided into different regions

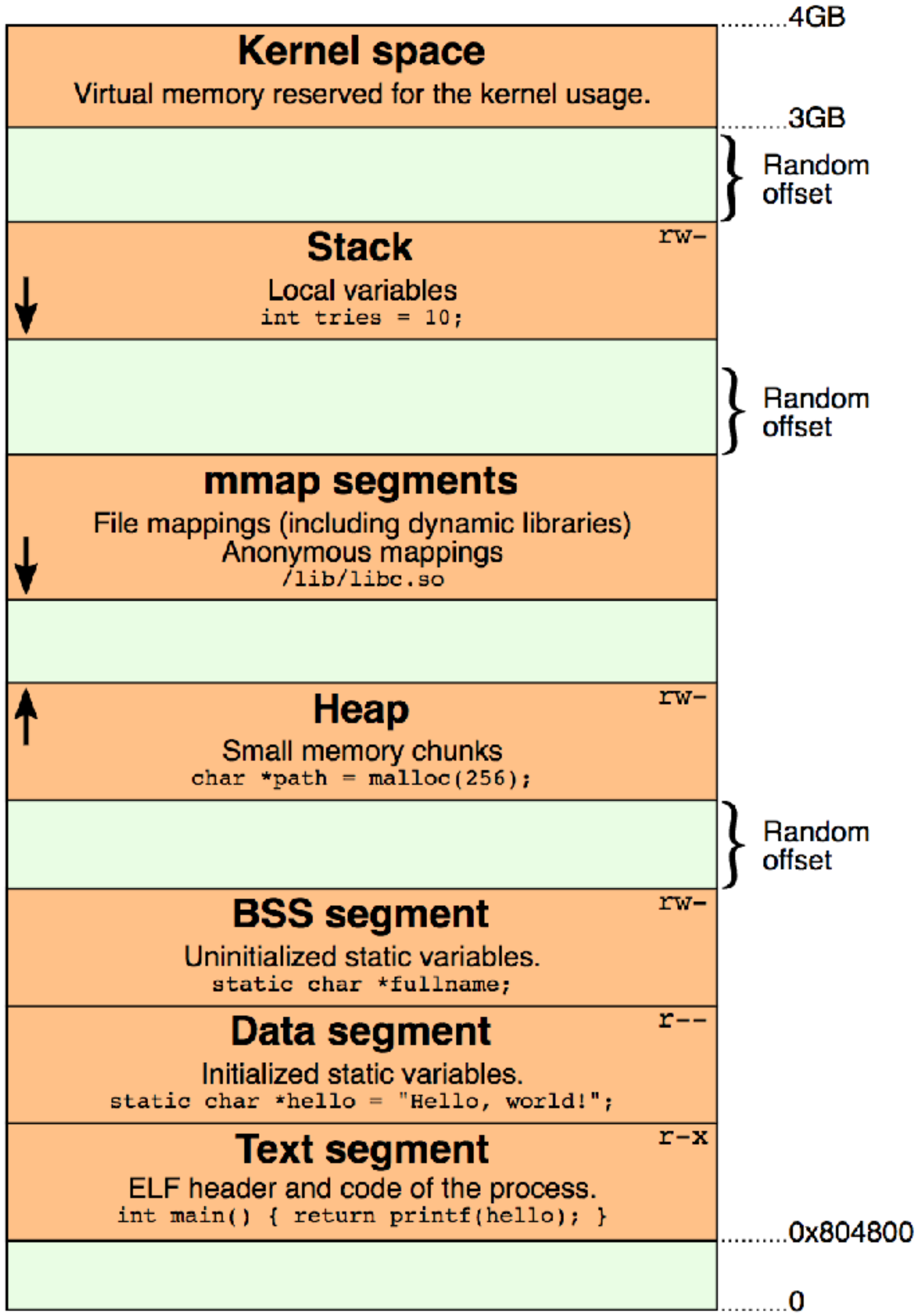
Name a few?

OS separates these into different **segments**



Kernel memory

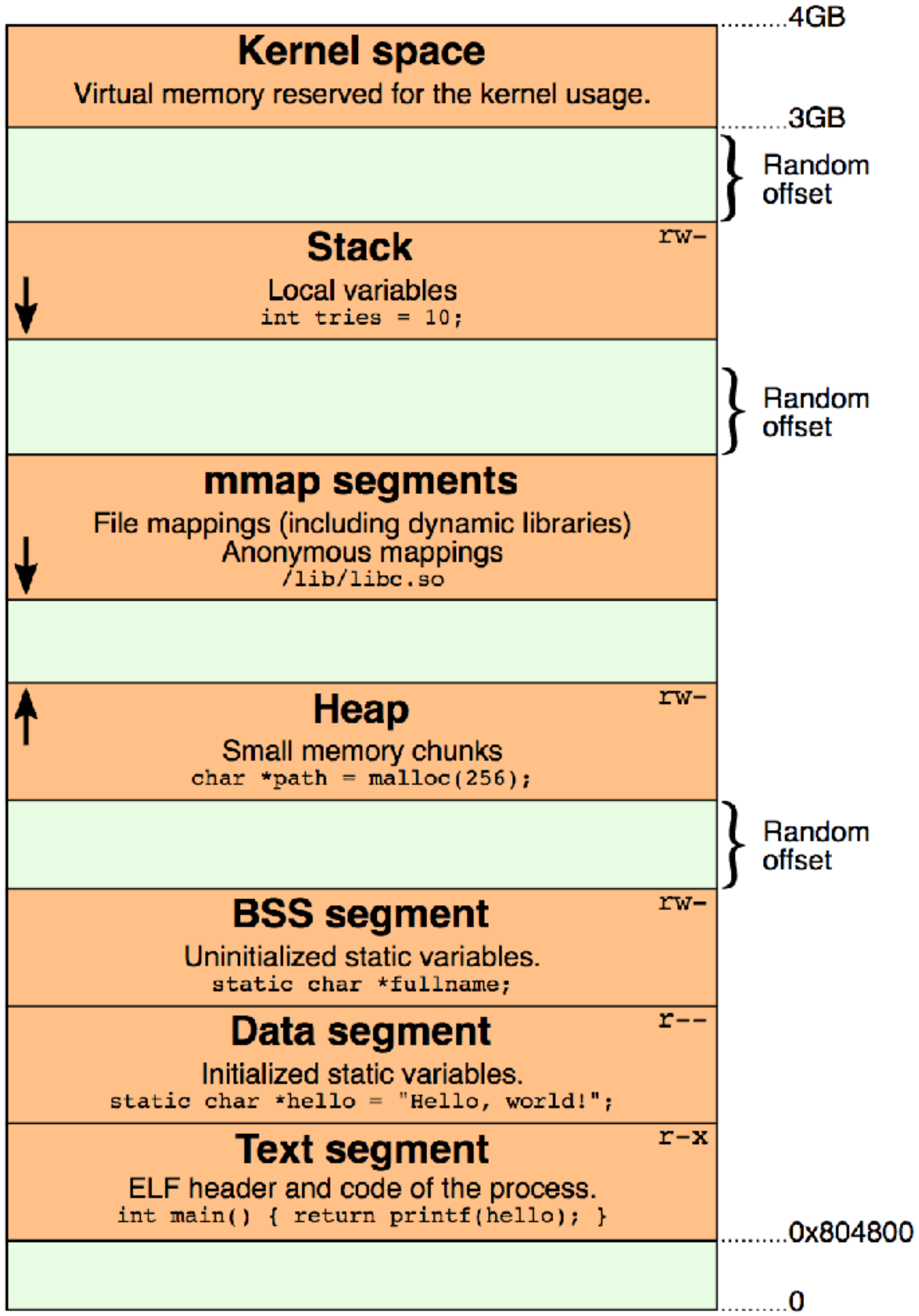
Your OS uses it



Stack: push / pop

Very important:

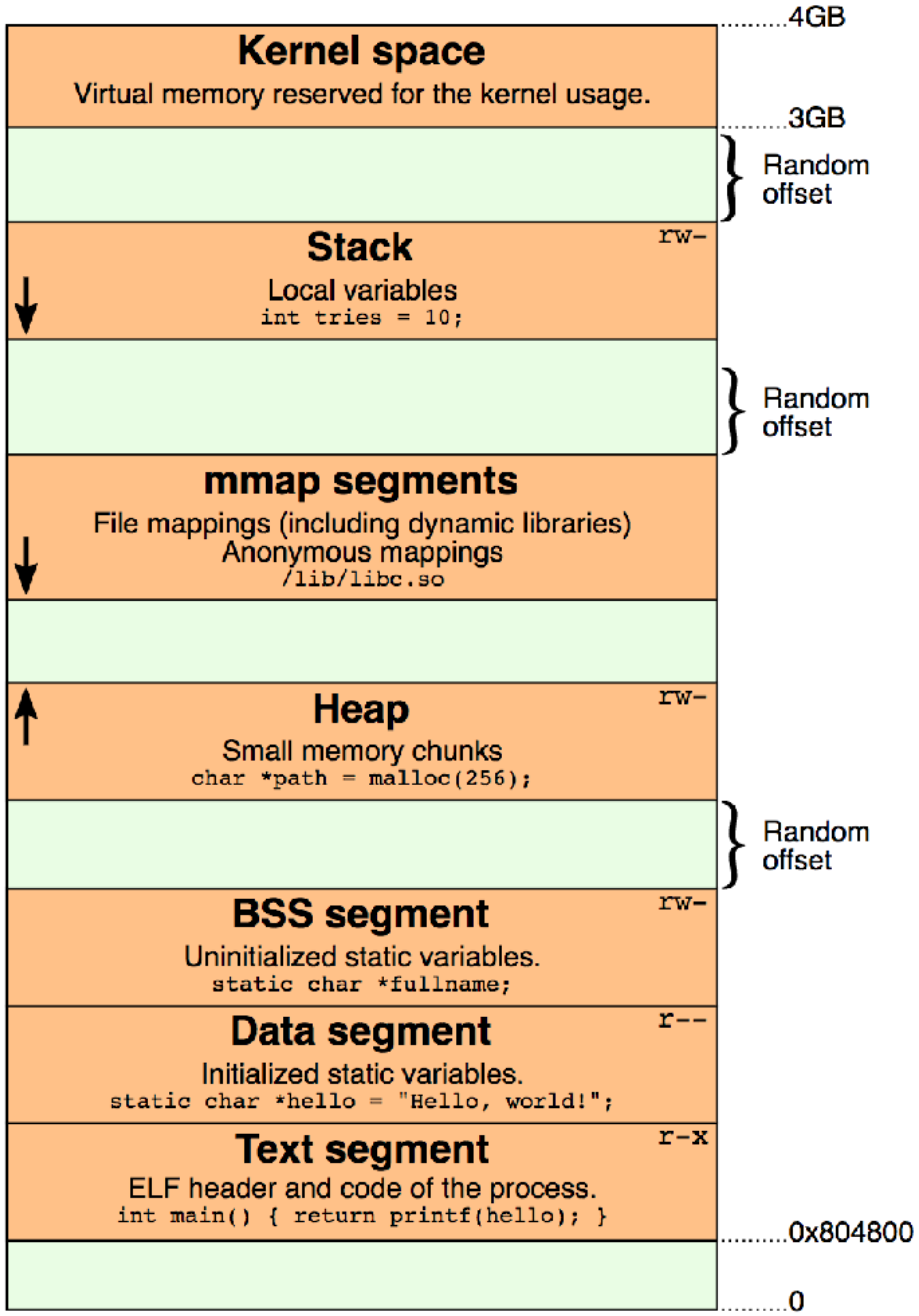
The stack grows **down**



Stack: push / pop

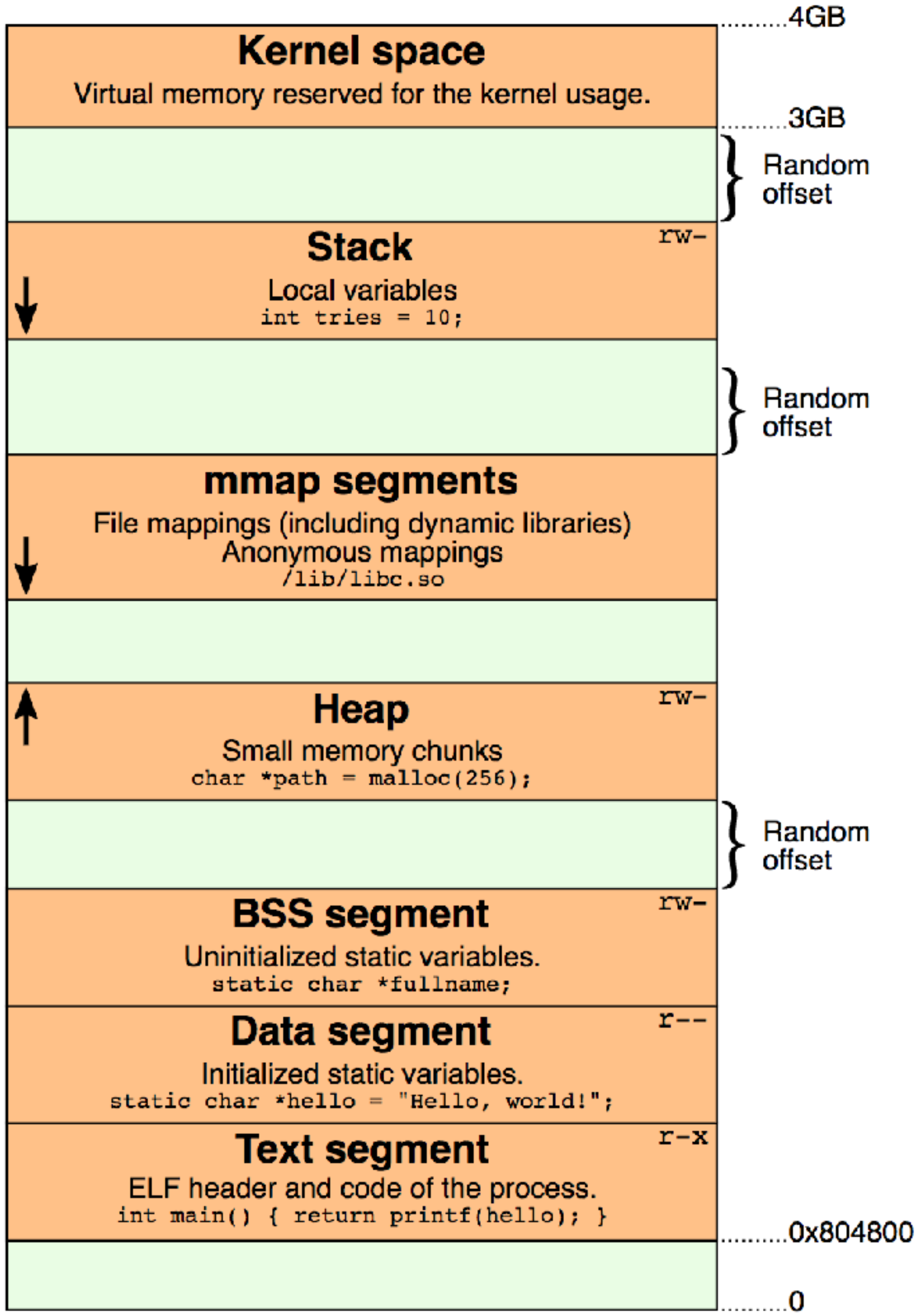
Very important:

The stack grows **down**



mmap segments

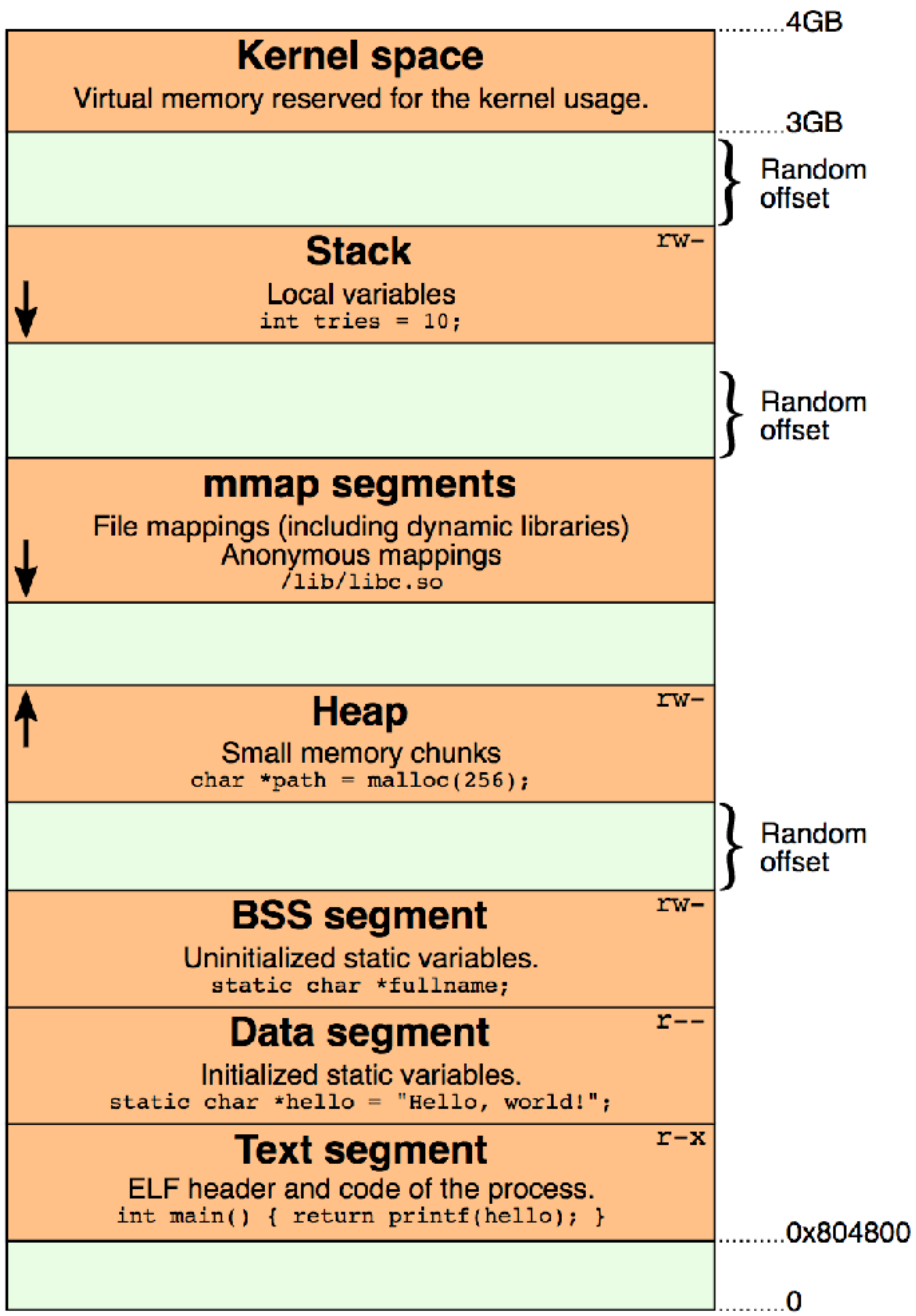
Allows you to **map** a file to memory



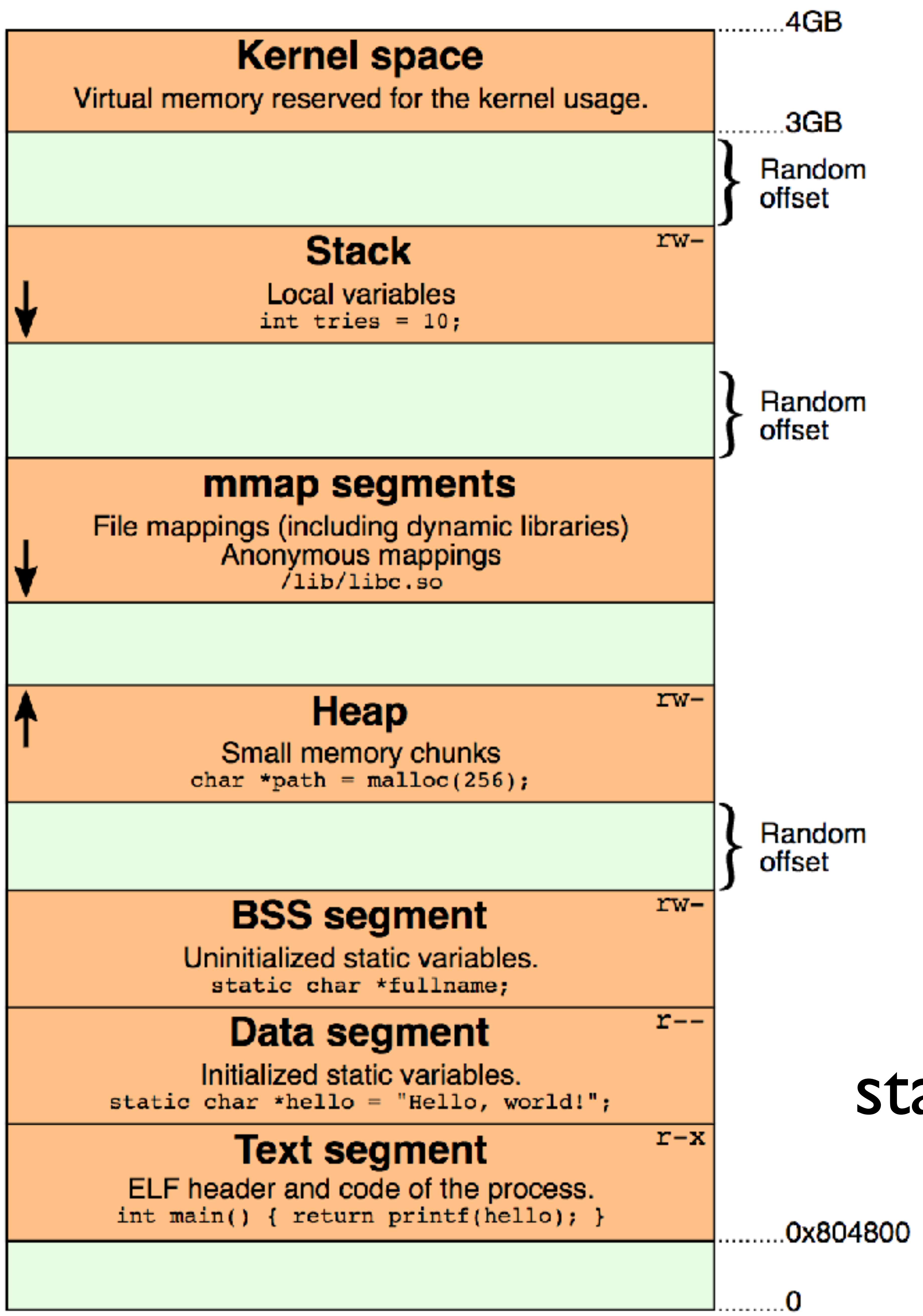
Heap: dynamic allocation

C++: New / delete

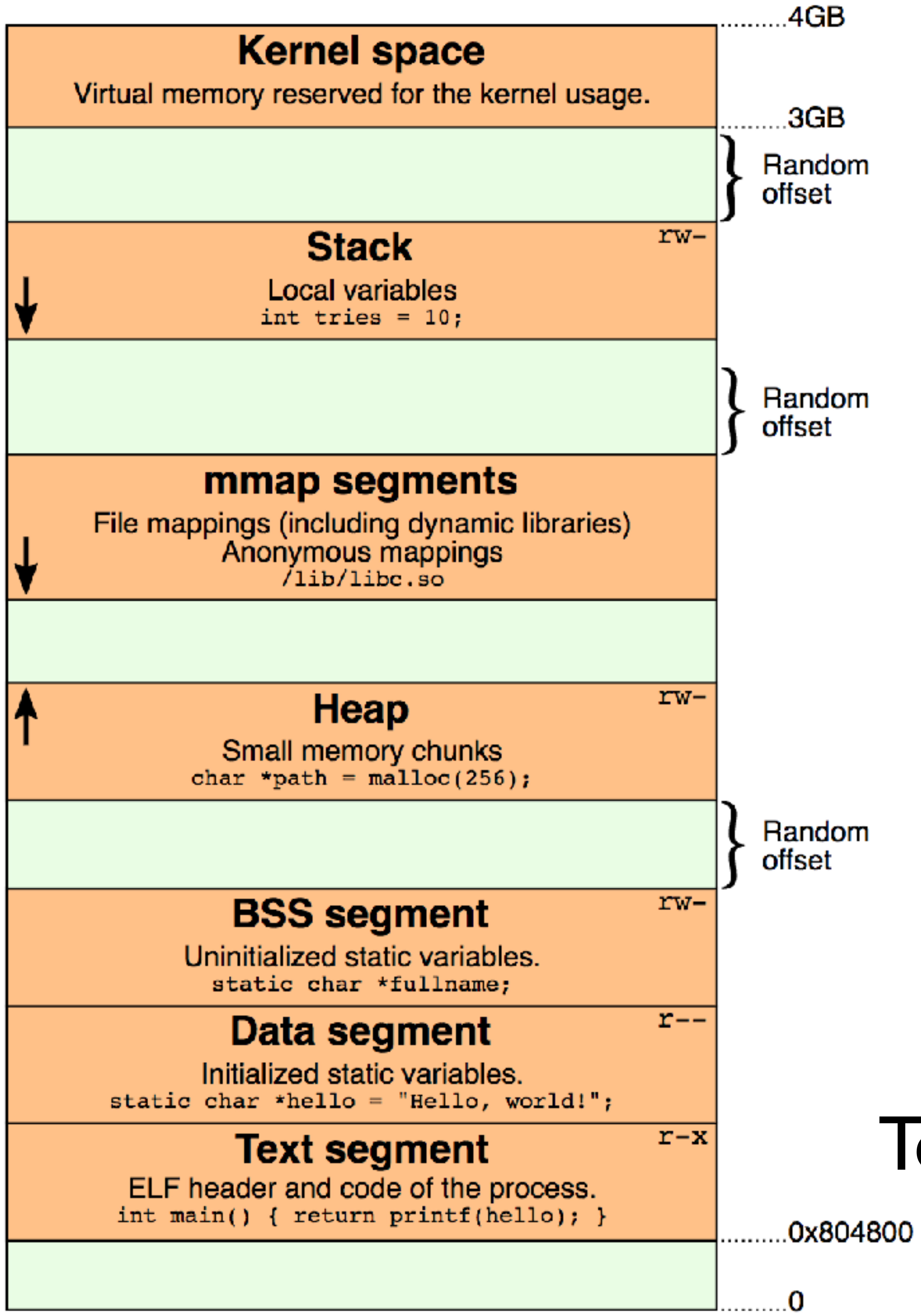
C: Malloc / free



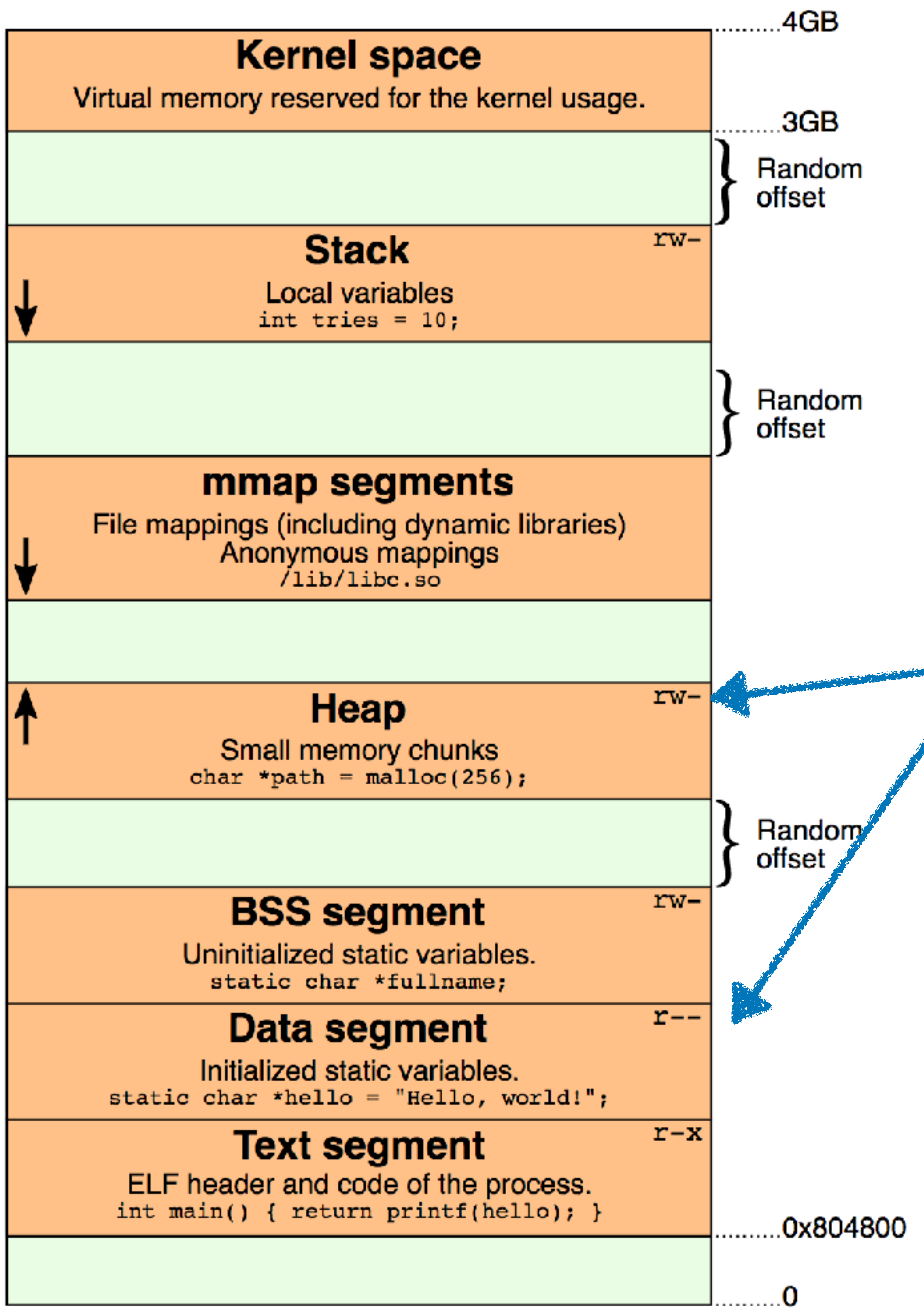
BSS: Uninitialized static vars (globals)



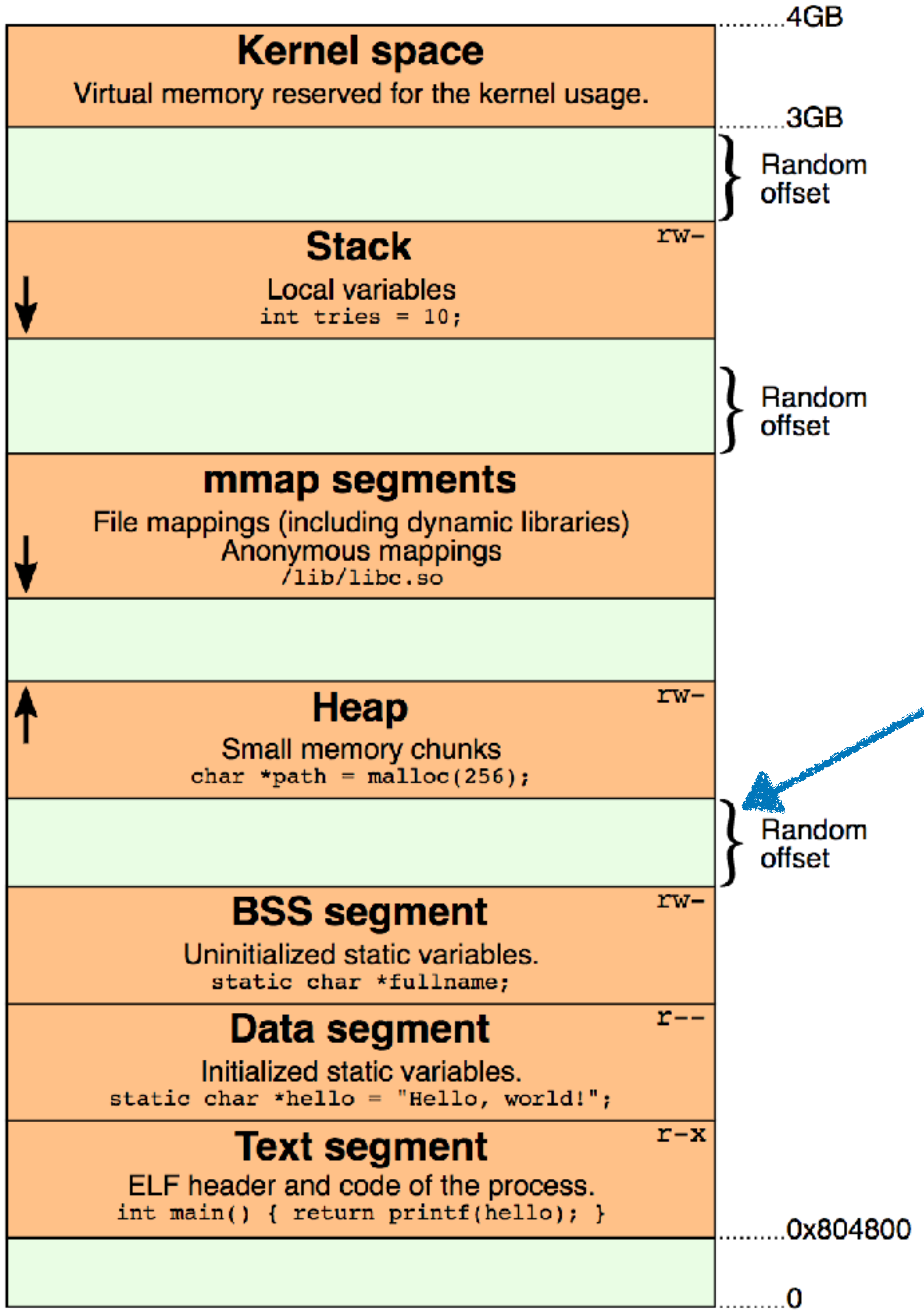
Data segment: initialized statics—e.g., constant strings



Text segment: program code



Note the **permissions**



This **random offset** really security feature

Calling conventions

Touch-tone phones, send an acoustic wave over the wire

If Alice wants to call Bob, her phone needs to send the right sounds over the wire in the right order

Calling conventions

When function A wants to call function B, it has to do the same

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Calling conventions

Modern computers use a few **different** calling conventions

De-facto standard (Linux / MacOS / etc..) : **x86-64 System V ABI**

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Note: this is **new** for the 64 bit API. You might see stuff online for the 32-bit API that is **different**

Calling conventions: x86-64 System V ABI

- Where do arguments go?
  - First six: rdi,rsi,rdx,rcx,r8,r9
- How to store return address?
  - `call` instruction puts on top of stack
- Who saves registers?
  - Caller saves caller-save registers
    - R10,R11, any ones used for args
- Where is result stored?
  - Result stored in %rax

x86-64 Integer Registers: Usage Conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

http://slideplayer.com/slide/9679824/

x86-64 System V ABI

Rules for **caller**:
- Save caller-save registers
- First six args in registers, after that put on stack
- Execute `call`—pushes ret addr

Afterwards:
- Pop saved registers
- Result now in %rax

x86-64 System V ABI

Rules for **callee**:

- First six args available in registers
- Push %rbp—caller's base pointer
- Move %rsp to %rbp—Setup new frame
- Subtract necessary stack space
- Push callee-save registers
- Before exit: restore rbp/callee-saved regs
  - `leave` instruction restores rbp
- When function done, put result in %rax
- Use `ret` instruction to pop return rip

These rules are cumbersome: I frequently look them up, they change depending on the kind of function you're calling, etc…

Upshot: don't feel you have to memorize, just get the gist / know how to recognize them

Small examples: interactive demo of x86-64 ABI

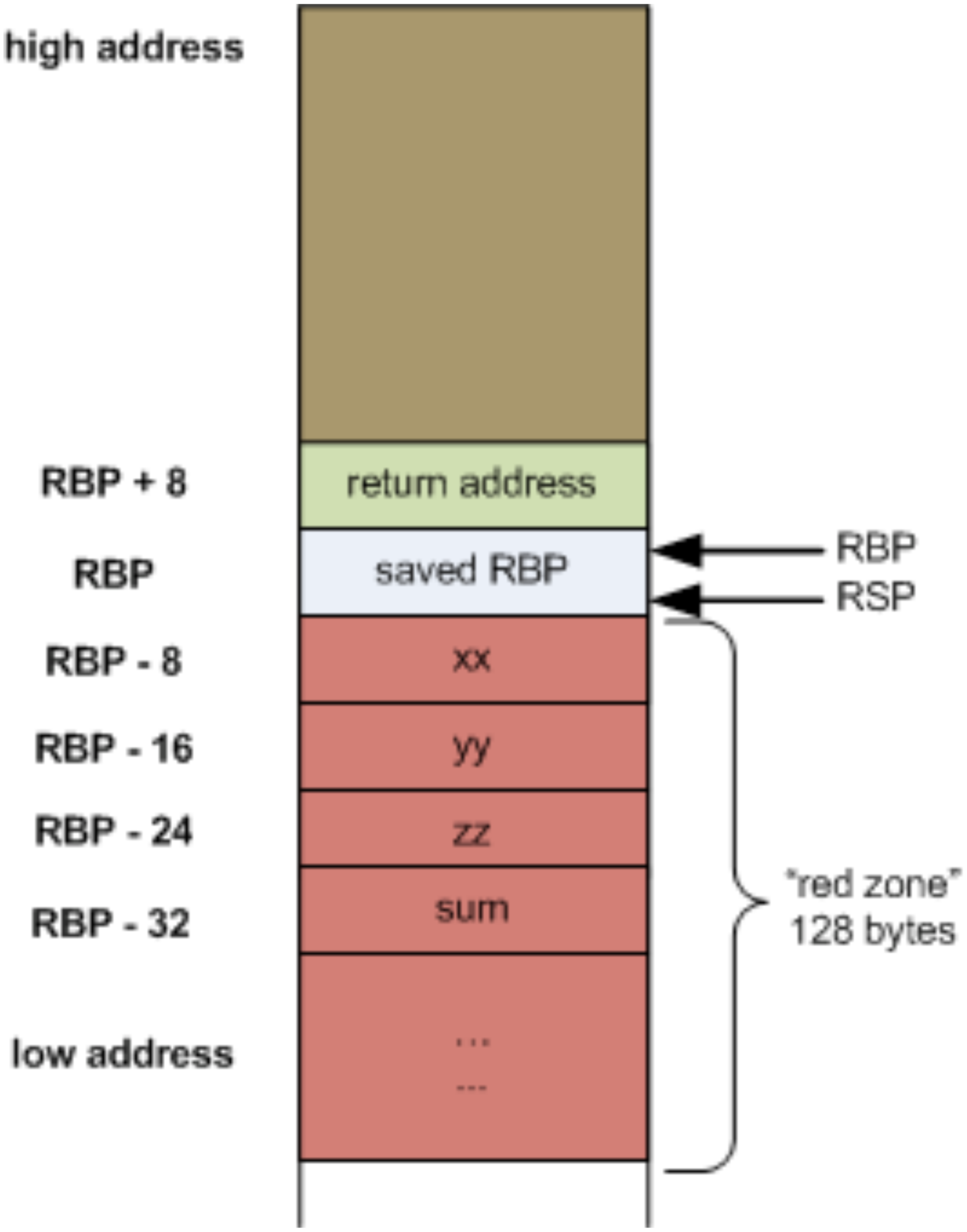
Trivia: the red zone

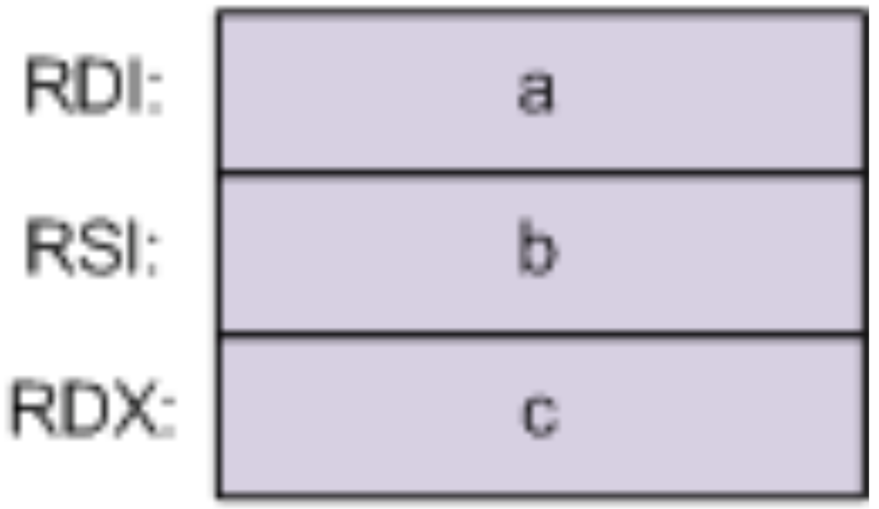
```
int bar(int a, int b) {
  return a + b;
}
```

Weird! This code using -4(%rbp) before decrementing the stack pointer!!

Turns out: x86-64 **guarantees** there are always128 bytes below %rsp

```
bar:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```





Upshot: if a function uses at most 128 bytes below RSP, doesn't have to subtract anything from RSP

This is an optimization for "small" functions: so they never have to subtract from RSP

Question: why does GCC generate such stupid code?

Answer: code unoptimized, add -O(1/2/3) to optimize it

-O0 generates code that is predictable and easy to read

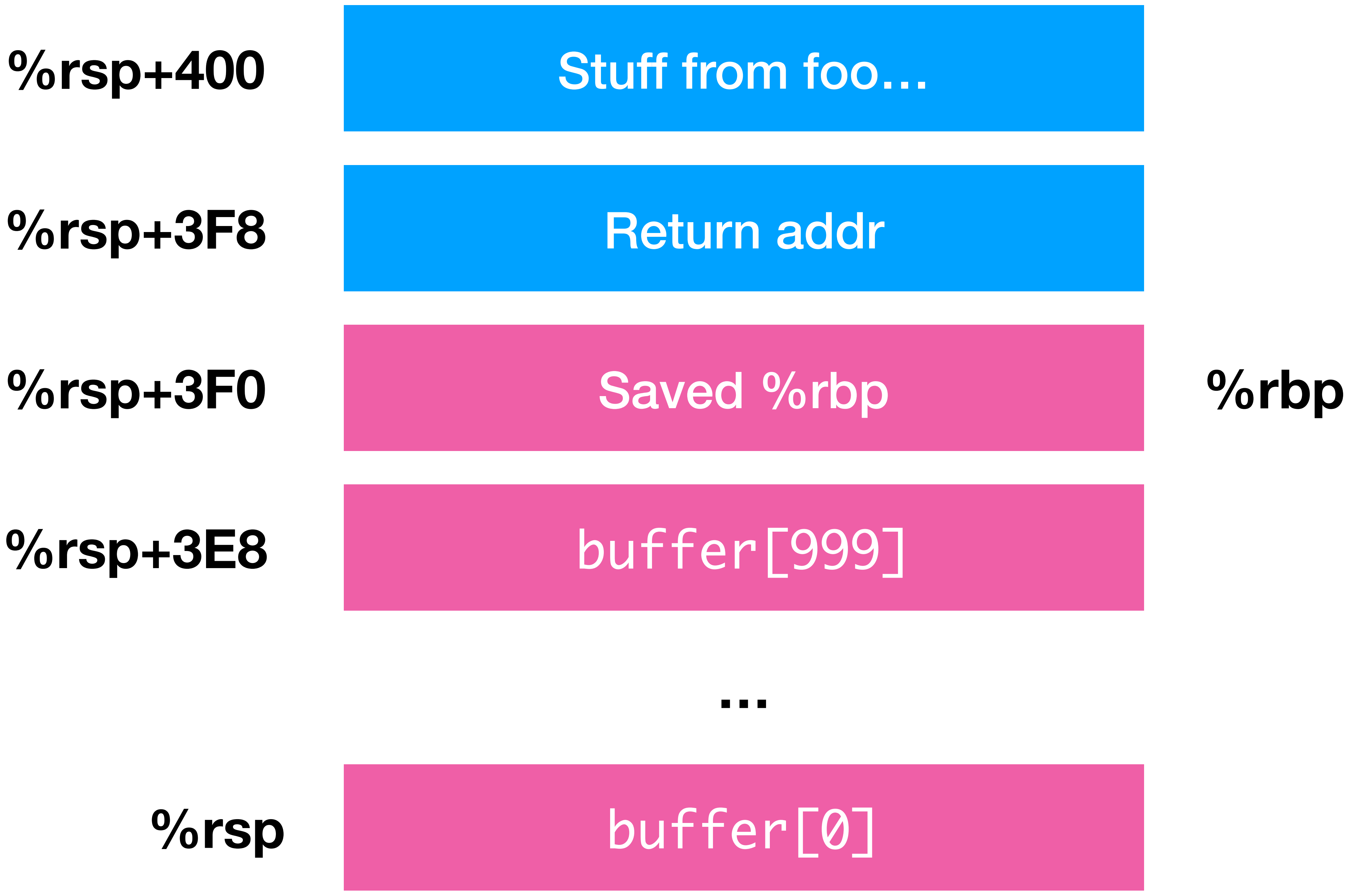

Coffee

First attack: Stack Smashing

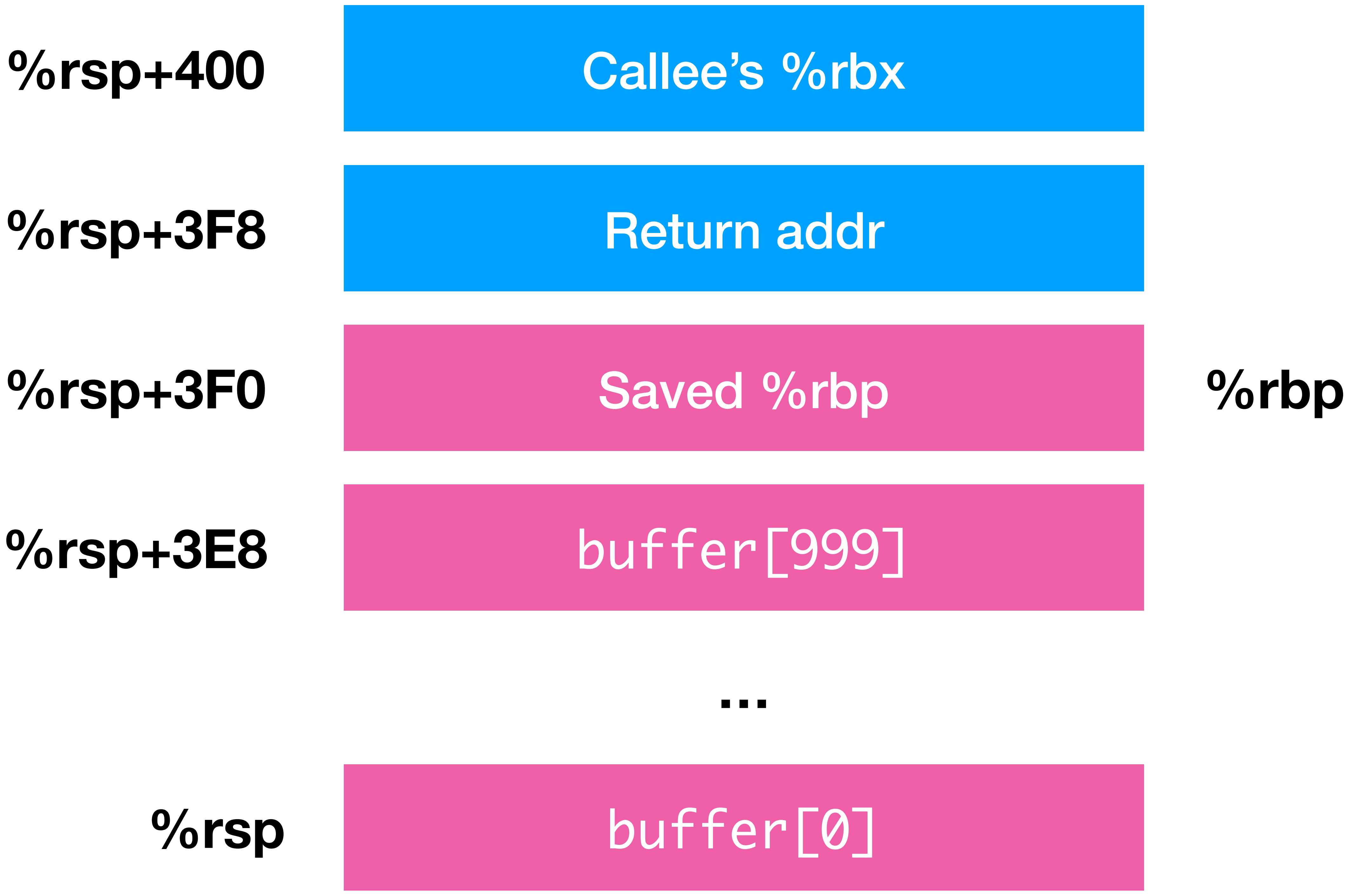
This code is bad because it doesn't check the length of the string in `ptr`…

```
void foo(char *ptr) {
    char buffer[1000];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

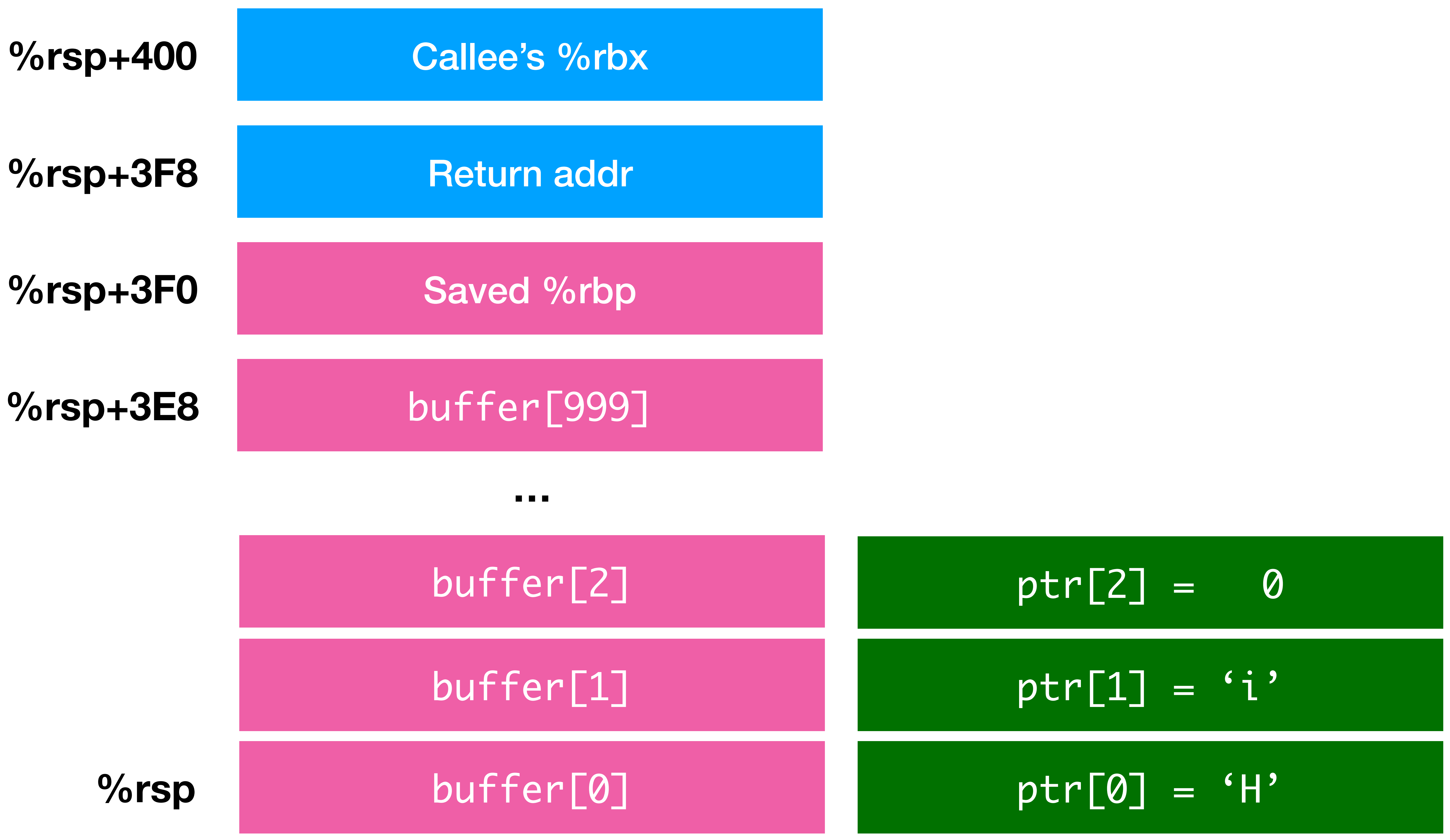
After foo starts



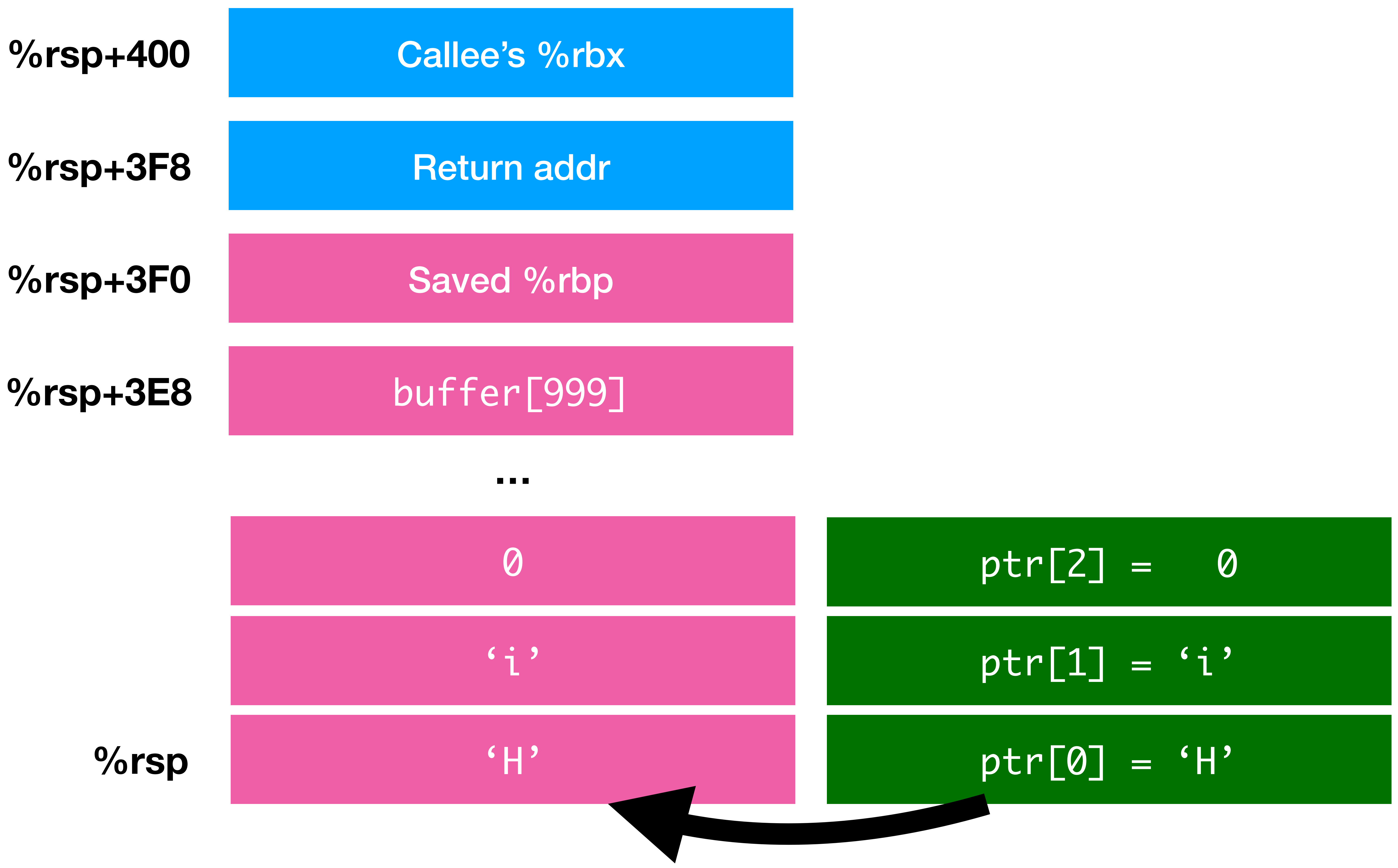
After foo starts



Key observation: the stack **grows down**



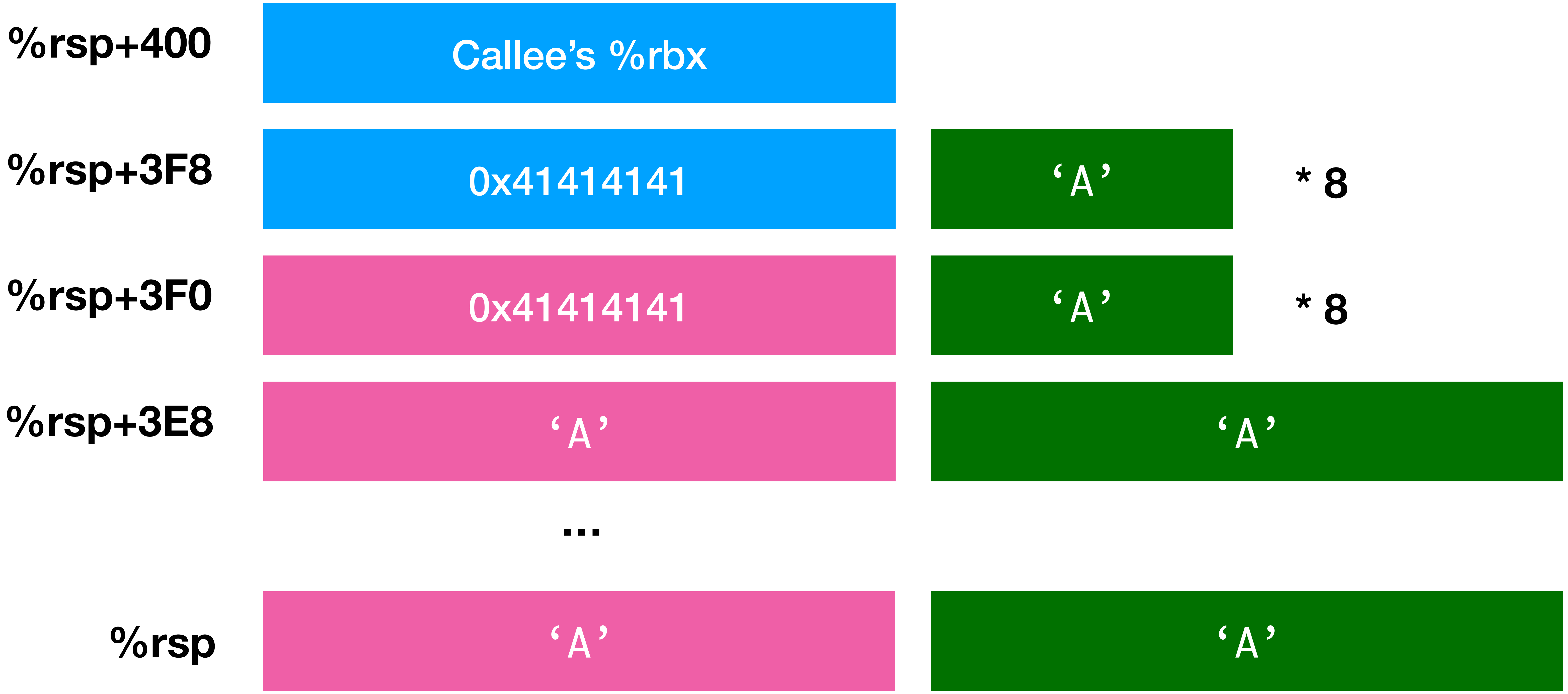
Consider what happens when `strcpy(buffer,ptr)`



Consider what happens when `strcpy(buffer,ptr)`

(This one is fine..)

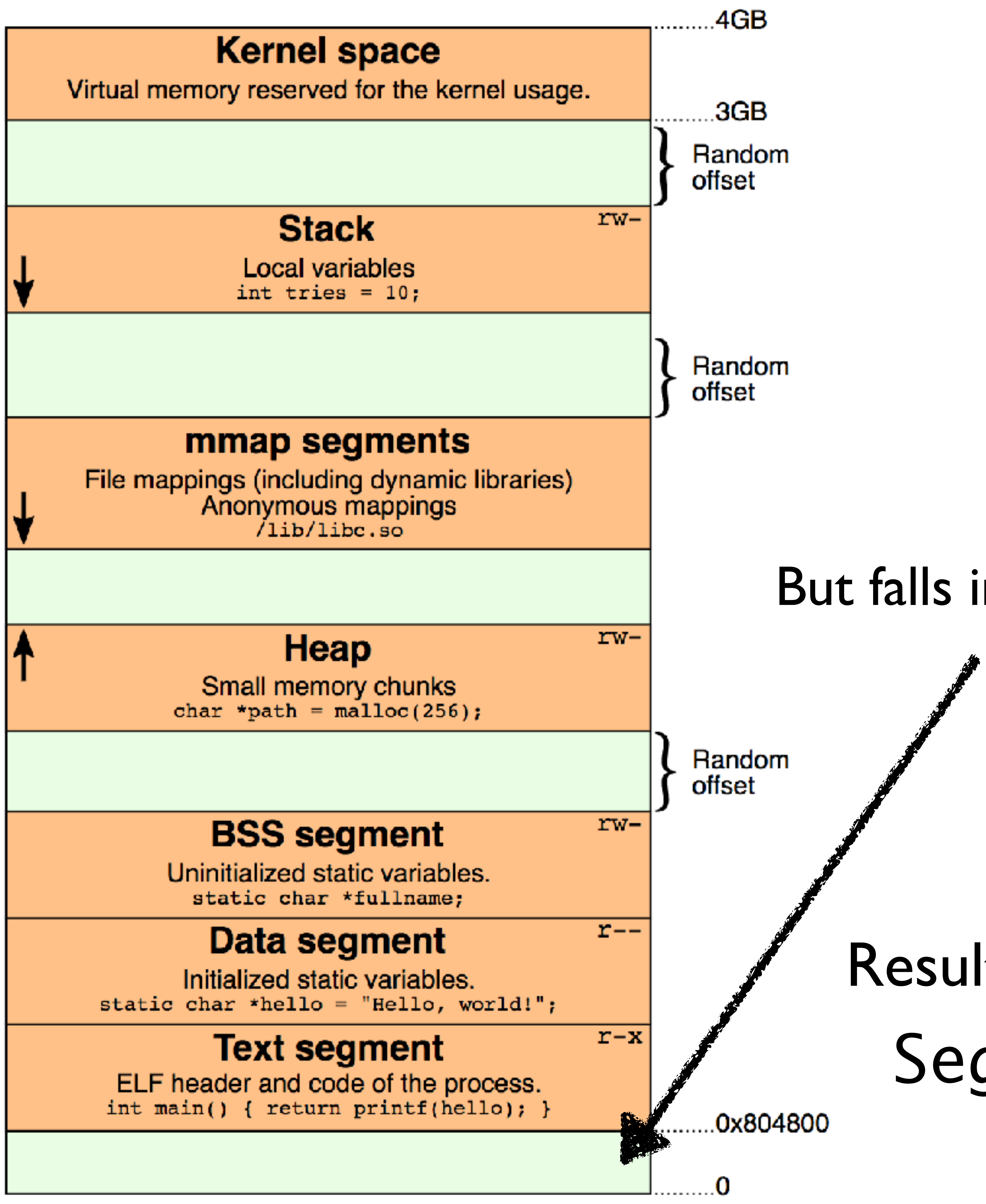
Now consider what happens when we provide input ‘A’ * 1008



Return addr becomes 0x41414141 ('A' four times)

Upon return, control goes to 0x41414141

If anything at this address, program will execute it



But falls in here, unmapped memory

Result: most common C crash

Segmentation Fault

The compiler translates binary code into machine code

```
execve("/bin/sh")
```

↓

Compiler

↓

We'll cover this assembly later in class!

```
"\x48\x31\xd2"                                 // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68" // mov   $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                             // shr    $0x8, %rbx
"\x53"                                         // push   %rbx
"\x48\x89\xe7"                                 // mov    %rsp, %rdi
"\x50"                                         // push   %rax
"\x57"                                         // push   %rdi
"\x48\x89\xe6"                                 // mov    %rsp, %rsi
"\xb0\x3b"                                     // mov    $0x3b, %al
"\x0f\x05";                                    // syscall
```

man execve

All that code is **loaded** by the kernel at a specific place in memory

Let’s assume for a second that the compiler loads that code at
`0x41414141`

In the next few slides we’ll see what happens if it’s **not** there

Return pointer: 0x41414141

After returning, we *expect* the code to go back here

```
// foo's caller
foo(p);
x = x+1;

void foo(char *ptr) {
    char buffer[ptr];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

0x41414141

```
"\x48\x31\xd2"                                  // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov   $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
"\x53"                                          // push   %rbx
"\x48\x89\xe7"                                  // mov    %rsp, %rdi
"\x50"                                          // push   %rax
"\x57"                                          // push   %rdi
"\x48\x89\xe6"                                  // mov    %rsp, %rsi
"\xb0\x3b"                                      // mov    $0x3b, %al
"\x0f\x05";                                     // syscall
```

Return pointer: 0x41414141

But the return address has been *overwritten* (stack has been smashed)

```
// foo’s caller
foo(p);
x = x+1;

void foo(char *ptr) {
    char buffer[ptr];
    strcpy(buffer, ptr);
    printf(“length: %d\n”, strlen(buffer));
}
```

Instead, return goes here

0x41414141

```
"\x48\x31\xd2"                                  // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov   $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
"\x53"                                          // push   %rbx
"\x48\x89\xe7"                                  // mov    %rsp, %rdi
"\x50"                                          // push   %rax
"\x57"                                          // push   %rdi
"\x48\x89\xe6"                                  // mov    %rsp, %rsi
"\xb0\x3b"                                      // mov    $0x3b, %al
"\x0f\x05";                                     // syscall
```

Now, the computer executes a shell instead!!!

Might not be so bad if it’s a local program

But bad if it’s a connection to a remote server!

In your first project, you’ll mount one of these attacks on a vulnerable file server

So my job **as an attacker** is to find a buffer overflow in the program
and then craft an input that sends the code where I want

Question 1: How do I find a bug?

A: Dig through the source manually, if source is available

(If source unavailable, use a ***decompiler***)

A: Some automated testing tools

2012 IEEE Symposium on Security and Privacy

Unleashing MAYHEM on Binary Code

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley
Carnegie Mellon University
Pittsburgh, PA
{*sangkilc, thanassis, alexandre.rebert, dbrumley*}*@cmu.edu*

***Abstract*—In this paper we present MAYHEM, a new system for automatically finding exploitable bugs in binary (i.e., executable) programs. Every bug reported by MAYHEM is accompanied by a working shell-spawning exploit. The working exploits ensure soundness and that each bug report is security-critical and actionable. MAYHEM works on raw binary code without debugging information. To make exploit generation possible at the binary-level, MAYHEM addresses two major technical challenges: actively managing execution paths without exhausting memory, and reasoning about *symbolic memory indices*, where a load or a store address depends on user input. To this end, we propose two novel techniques: 1) hybrid symbolic execution for combining online and offline (concolic) execution to maximize the benefits of both techniques, and 2) index-based memory modeling, a technique that allows MAYHEM to efficiently reason about symbolic memory at the binary level. We used MAYHEM to find and demonstrate 29 exploitable vulnerabilities in both Linux and Windows programs, 2 of which were previously undocumented.**

***Keywords*-hybrid execution, symbolic memory, index-based memory modeling, exploit generation**

I. INTRODUCTION

Bugs are plentiful. For example, the Ubuntu Linux bug management database currently lists over 90,000 open bugs [17]. However, bugs that can be exploited by attackers are typically the most serious, and should be patched first. Thus, a central question is not whether a program has bugs, but which bugs are exploitable.

In this paper we present MAYHEM, a sound system for automatically finding exploitable bugs in binary (i.e., executable) programs. MAYHEM produces a working control-

In order to tackle this problem, MAYHEM's design is based on four main principles: 1) the system should be able to make forward progress for arbitrarily long times—ideally run "forever"—without exceeding the given resources (especially memory), 2) in order to maximize performance, the system should not repeat work, 3) the system should not throw away any work—previous analysis results of the system should be reusable on subsequent runs, and 4) the system should be able to reason about symbolic memory where a load or store address depends on user input. Handling memory addresses is essential to exploit real-world bugs. Principle #1 is necessary for running complex applications, since most non-trivial programs will contain a potentially infinite number of paths to explore.

Current approaches to symbolic execution, e.g., CUTE [26], BitBlaze [5], KLEE [9], SAGE [13], McVeto [27], AEG [2], S2E [28], and others [3], [21], do not satisfy all the above design points. Conceptually, current executors can be divided into two main categories: offline executors — which concretely run a single execution path and then symbolically execute it (also known as trace-based or *concolic* executors, e.g., SAGE), and online executors — which try to execute all possible paths in a single run of the system (e.g., S2E). Neither online nor offline executors satisfy principles #1-#3. In addition, most symbolic execution engines do not reason about symbolic memory, thus do not meet principle #4.

Offline symbolic executors [5], [13] reason about a single execution path at a time. Principle #1 is satisfied by iteratively picking new paths to explore. Further, every run of the

We're launching an angr blog! The first post, with plans for the upcoming year, is here.

What is angr?

angr is a python framework for analyzing binaries. It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks.

As an introduction to angr's capabilities, here are some of the things that you can do using angr and the tools built with it:

- Control-flow graph recovery. ***show code***
- Symbolic execution. ***show code***
- Automatic ROP chain building using angrop. ***show code***
- Automatically binaries hardening using patcherex. ***show code***
- Automatic exploit generation (for DECREE and simple Linux binaries) using rex. ***show code***
- Use angr-management, a (very alpha state!) GUI for angr, to analyze binaries! ***show code***
- Achieve cyber-autonomy in the comfort of your own home, using Mechanical Phish, the third-place winner of the DARPA Cyber Grand Challenge.

angr itself is made up of several subprojects, all of which can be used separately in other projects:

- an executable and library loader, CLE
- a library describing various architectures, archinfo
- a Python wrapper around the binary code lifter VEX, PyVEX
- a data backend to abstract away differences between static and symbolic domains, Claripy
- the program analysis suite itself, angr

How do I learn?

There are a few resources you can use to help you get up to speed!

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 2: What if program doesn't have bugs!?

A: You're hosed, can't perform this attack

But some other attacks we'll talk about on Thursday

The best way to prevent these attacks is to write in languages where these bugs can't occur!!

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 3: How do I know what code to execute?

A: Find the code you want in the binary

A: We'll also learn how you can **inject your own** code

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 4: How do I know **where the code is**

A: Use GDB to find it after booting up the binary

But there's a critical catch!

The compiler includes a variety of **protections** against stack smashing

Stack canaries (which we'll learn about next week)

Address **S**pace **L**ayout **R**andomization

Loads code into random addr each run!

(We'll see some techniques to help defeat this)