

How Software Executes

CS-576 Systems Security

Instructor: Georgios Portokalidis

Spring 2018

Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

Programming Languages

C, C++

Java, C#

Python, Perl, PHP

C, C++

Compiles to machine code

Typed but weakly enforced

Low-level memory access

User manages memory

Java, C#

Java

- Compiles to bytecode, run by the Java virtual machine
 - Initially interpreted, quickly just-in-time translated

C#

- Mix of compile and JIT

Type safe and strongly typed

Automatic memory management

Implicit memory access

Python, Perl, PHP

Dynamically typed (duck type)

- Types are checked for suitability at run time

Strong typed

- Operations are checked for safety

Interpreted

- PHP now also uses JIT

Automatic memory management

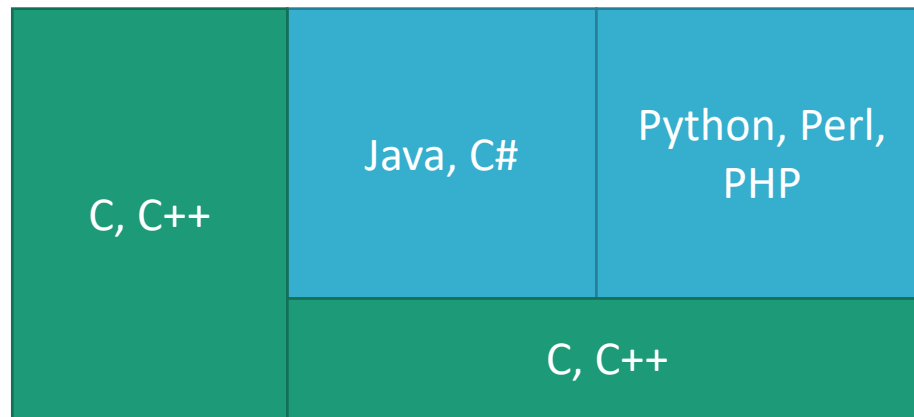
Why Do I Care?

What happens:

- When you have the following array: `char buffer[4];`
- And the following statement: `buffer[4] = 'A';`

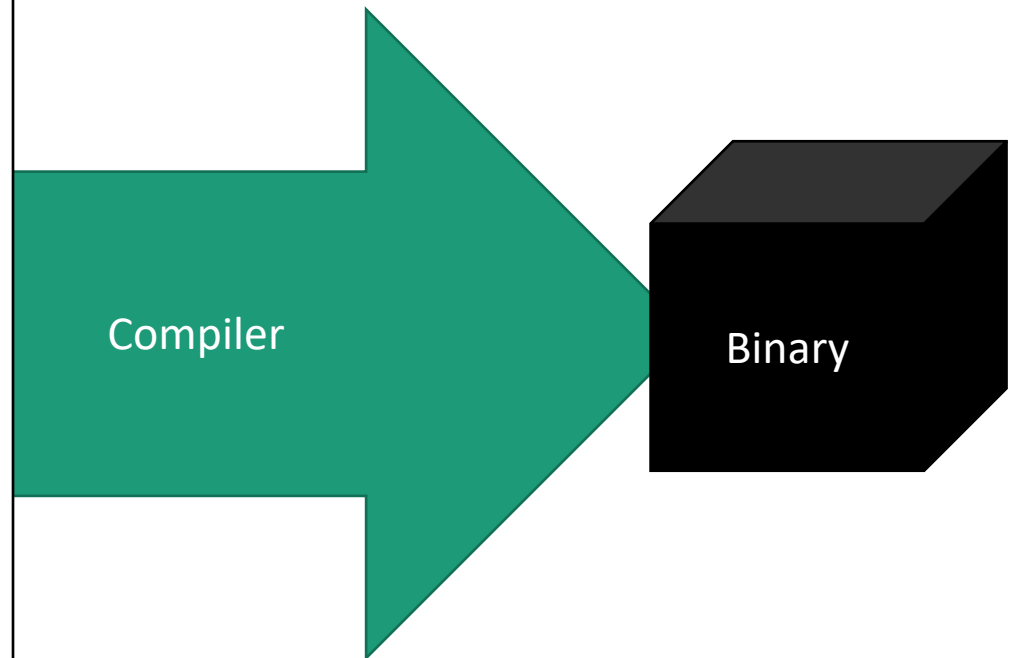
Whatever you guessed may be correct!

It's All C In the End



Source to Binary

```
static int total;  
static int total_over_threshold = 0;  
static int arbitrary_theshold = 16700;  
  
int simple_function(int a, int b, int x)  
{  
    int y;  
  
    y = a*x*x + b*x;  
  
    if (y > arbitrary_theshold)  
        total_over_threshold++;  
  
    total++;  
  
    return total;  
}
```



Compilation Process

Multiple stages

Usual compiler components

- Front-end parses source code into an internal representation (IR)
- Plenty of optimizations applied on the IR
- Machine code generation back-end generates binary code

Compiler IR

Appropriate for performing optimizations

Easy to map to different machine architectures

For example, the LLVM compiler models a machine with an infinite number of registers

```
define i32 @simple_function(i32 %a, i32 %b, i32 %x) #0 {  
    %1 = alloca i32, align 4  
    %2 = alloca i32, align 4  
    %3 = alloca i32, align 4  
    %y = alloca i32, align 4  
    store i32 %a, i32* %1, align 4  
    store i32 %b, i32* %2, align 4  
    store i32 %x, i32* %3, align 4  
    %4 = load i32* %1, align 4  
    %5 = load i32* %3, align 4  
    %6 = mul nsw i32 %4, %5  
    %7 = load i32* %3, align 4  
    %8 = mul nsw i32 %6, %7  
    %9 = load i32* %2, align 4  
    %10 = load i32* %3, align 4  
    %11 = mul nsw i32 %9, %10  
    %12 = add nsw i32 %8, %11  
    store i32 %12, i32* %y, align 4  
    %13 = load i32* %y, align 4  
    %14 = load i32* @arbitrary_theshold, align 4  
    %15 = icmp sgt i32 %13, %14  
    br i1 %15, label %16, label %19  
; <label>:16                                ; preds = %0  
    %17 = load i32* @total_over_threshold, align 4  
    %18 = add nsw i32 %17, 1  
    store i32 %18, i32* @total_over_threshold,  
    align 4  
}
```

Assembly Code

Assembly code is not binary

Low(est) level code

The language corresponds
to the actual machine
instructions that hardware
can execute

AT&T syntax: instr src, dest

Intel syntax: instr dst, src

simple_function:

GAS, AT&T syntax

```
pushq %rbp
movq  %rsp, %rbp
movl  %edi, -20(%rbp)
movl  %esi, -24(%rbp)
movl  %edx, -28(%rbp)
movl  -20(%rbp), %eax
imull -28(%rbp), %eax
movl  %eax, %edx
movl  -24(%rbp), %eax
addl  %edx, %eax
imull -28(%rbp), %eax
movl  %eax, -4(%rbp)
movl  arbitrary_theshold(%rip),
```

%eax

```
cmpl  %eax, -4(%rbp)
```

```
jle  .L2
```

```
movl
```

```
total_over_threshold(%rip), %eax
```

```
addl  $1, %eax
```

```
movl  %eax,
```

```
total_over_threshold(%rip)
```

```
.L2:
```

```
movl  total(%rip), %eax
```

Programming Models

High-level languages

Variables

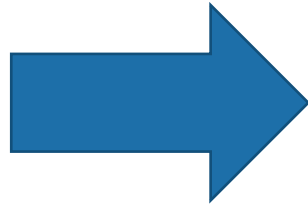
Routines

Objects

APIs

Libraries

...



Assembly

Registers

Flat memory model

Routines

Stack

...

How the World Really Is

PC: Program counter

- Address of next instruction

Register file

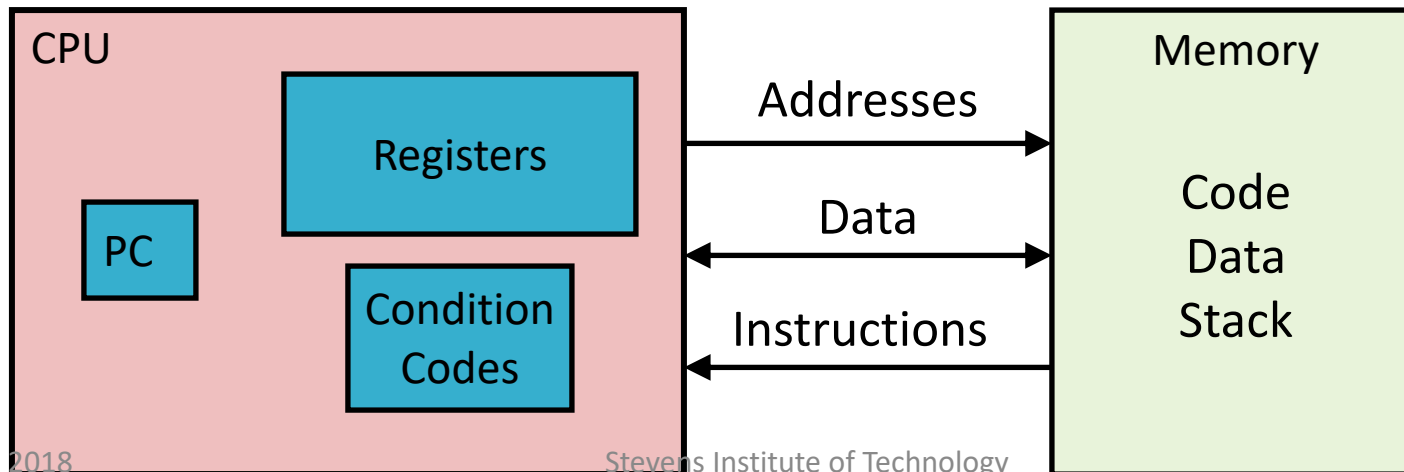
- Heavily used program data

Condition codes

- Store status information about most recent arithmetic or logical operation
- Used for conditional branching

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures



Overview

Introduction

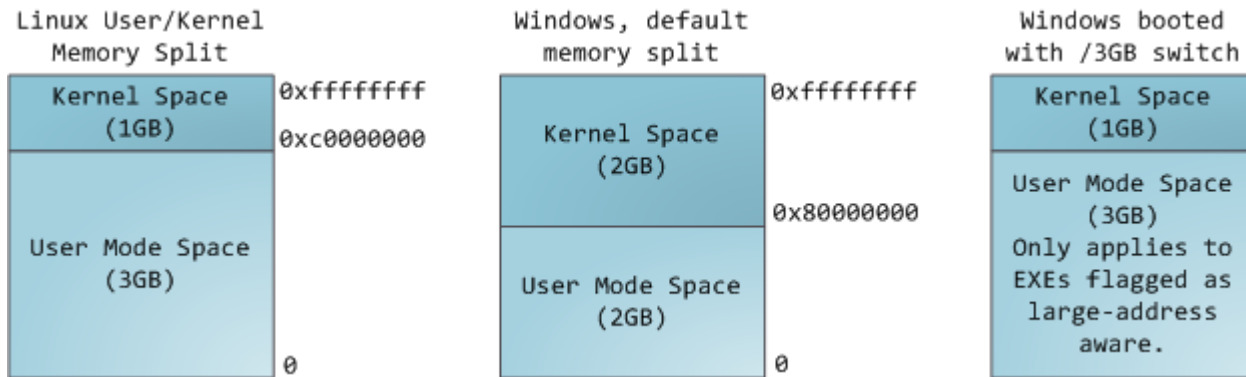
Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

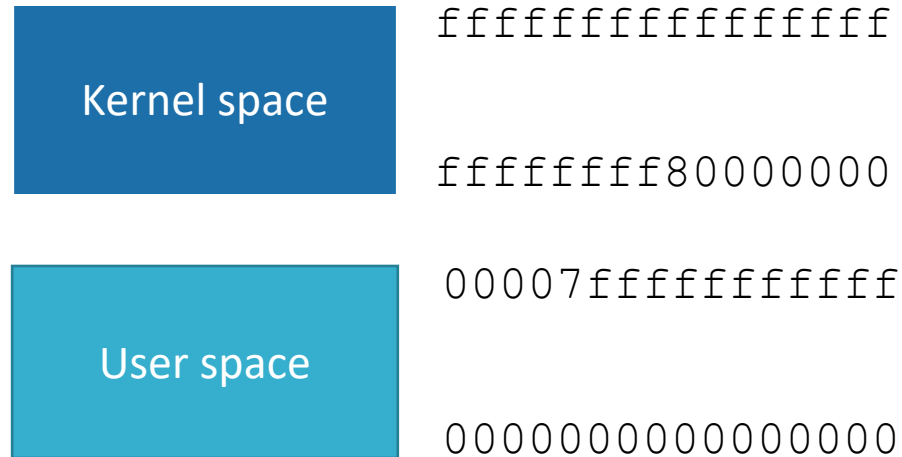
Anatomy of a Program in Memory



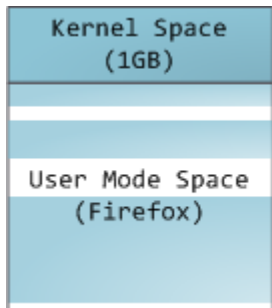
32-bit OS

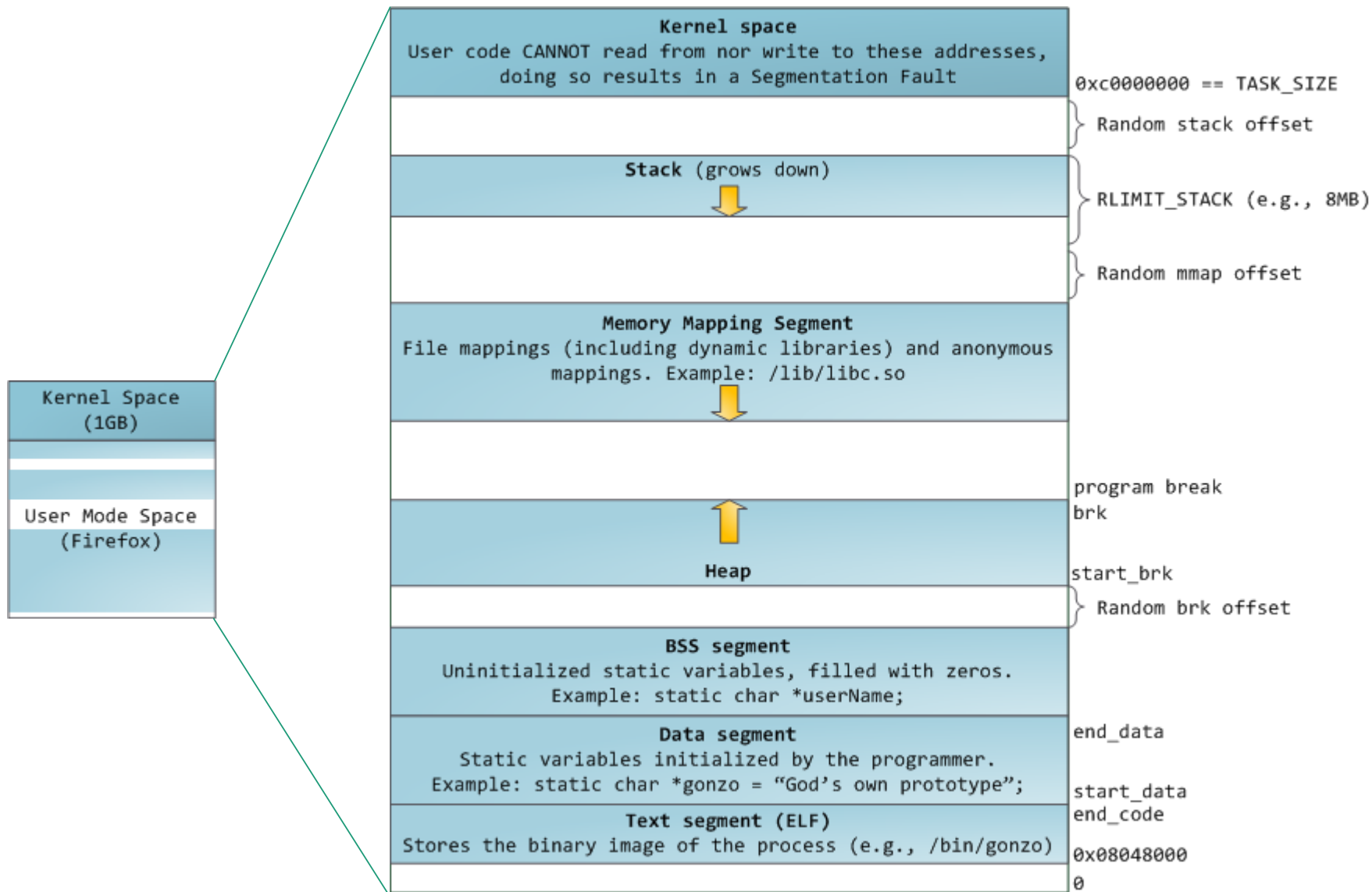
<https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>

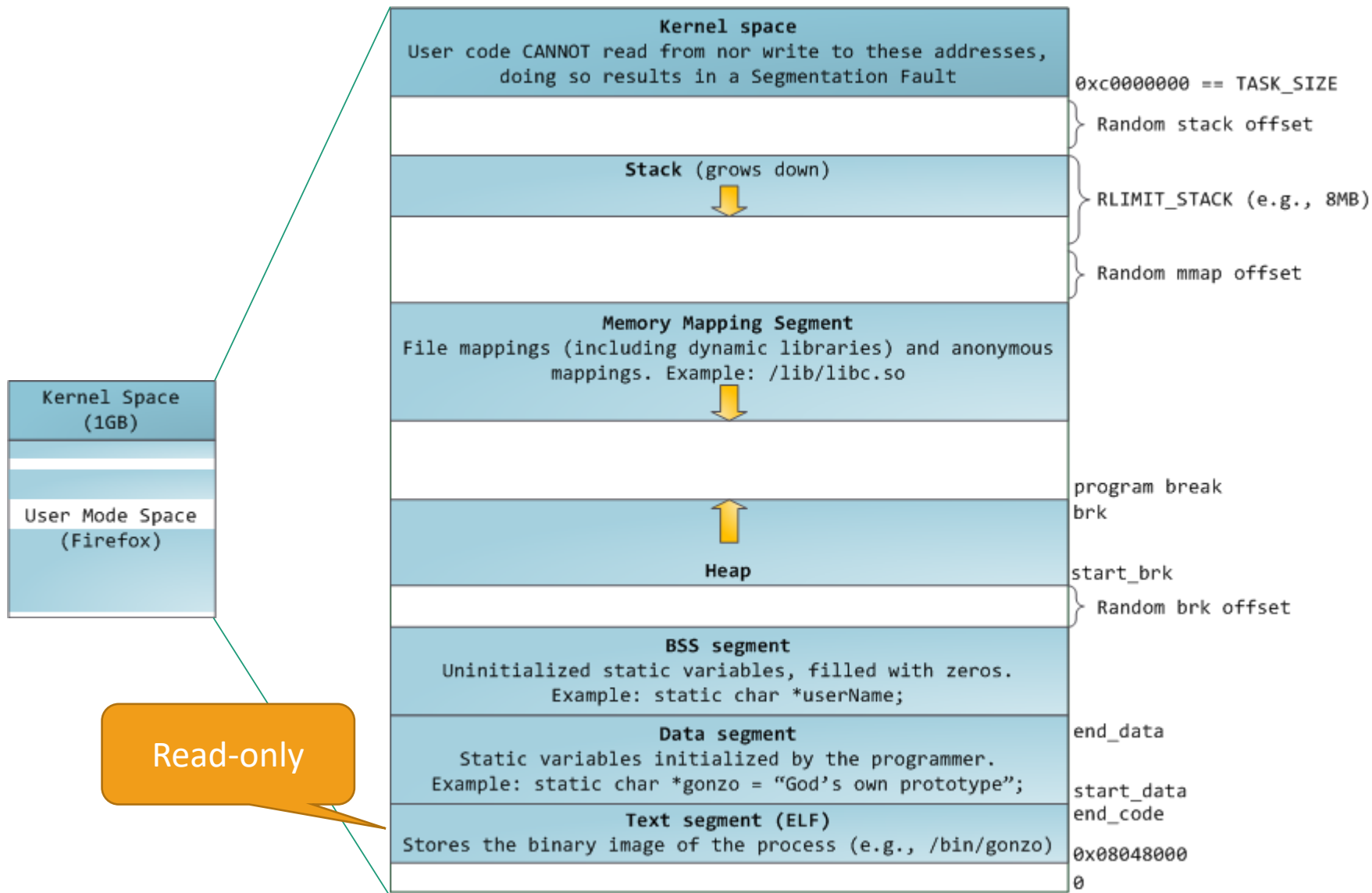
User/Kernel Space on 64-bit

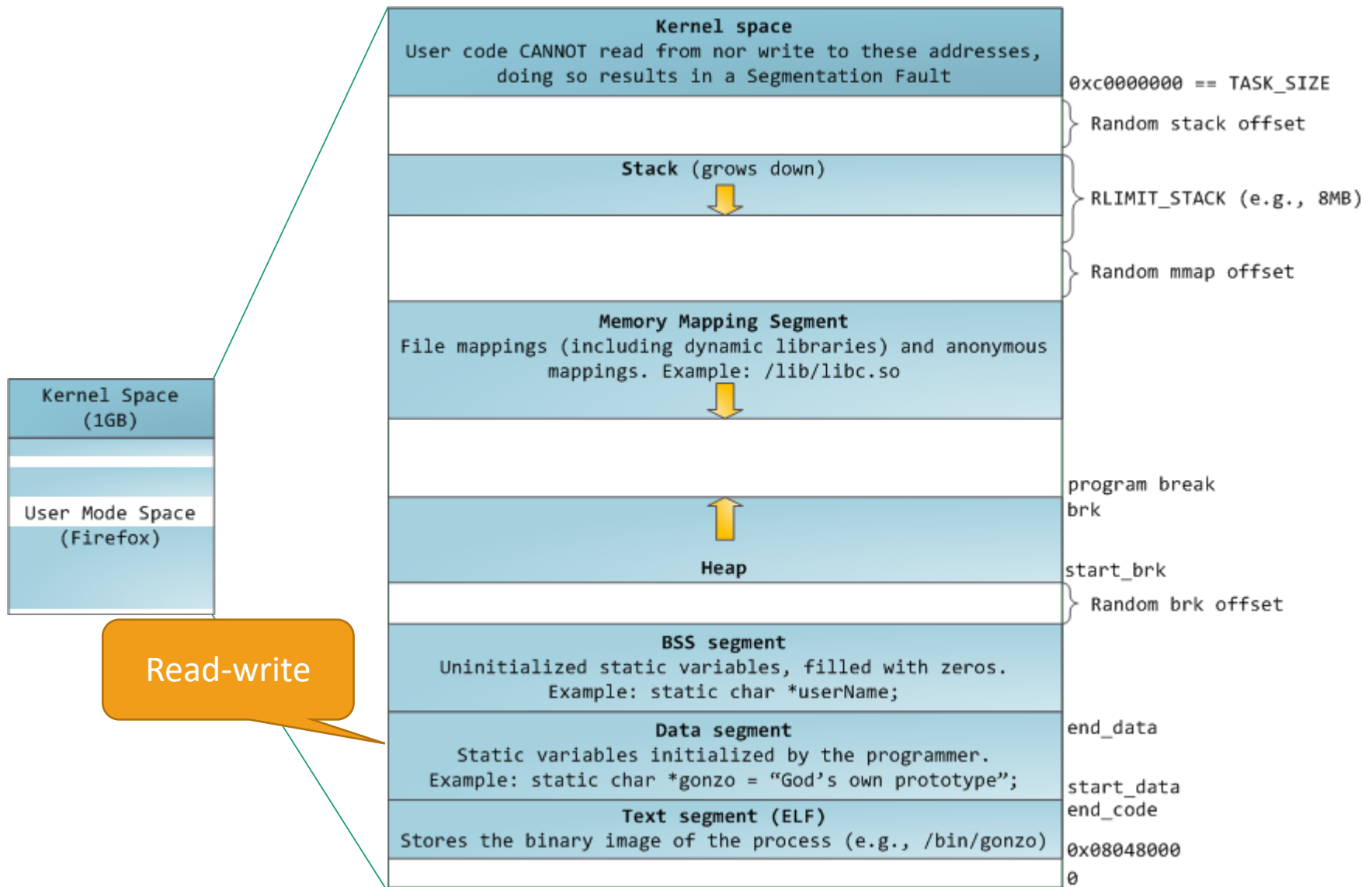


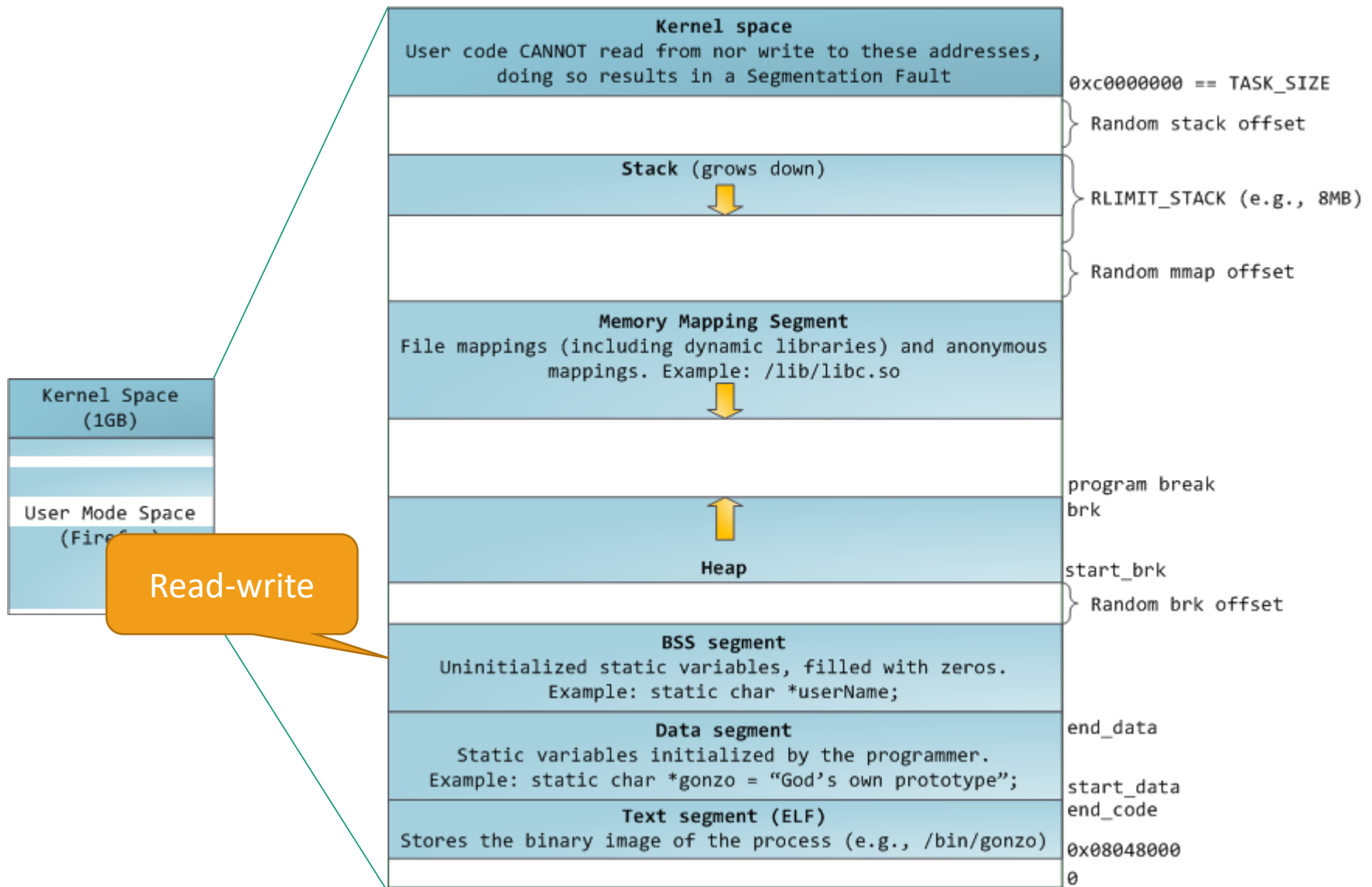
Process Anatomy



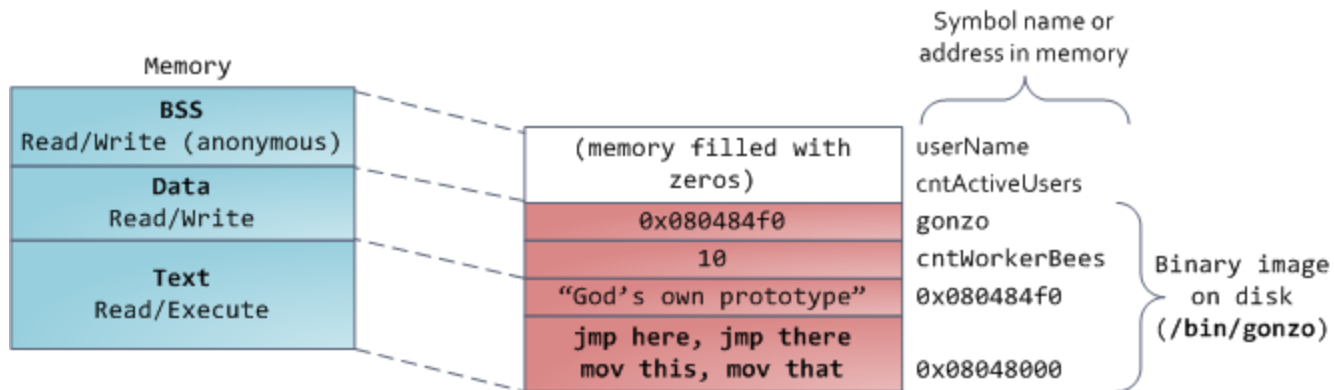








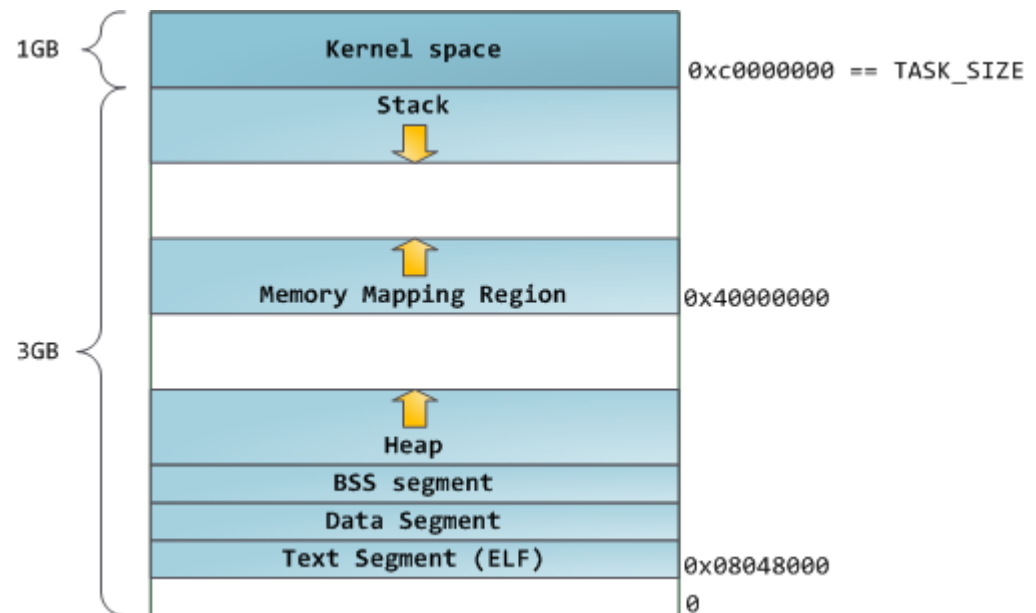
Memory vs Disk



Stack

Contains arguments, local variables, function return addresses

Read-write



Heap

Contains dynamically allocated data

Allocated by the OS using `brk()` and `sbrk()`

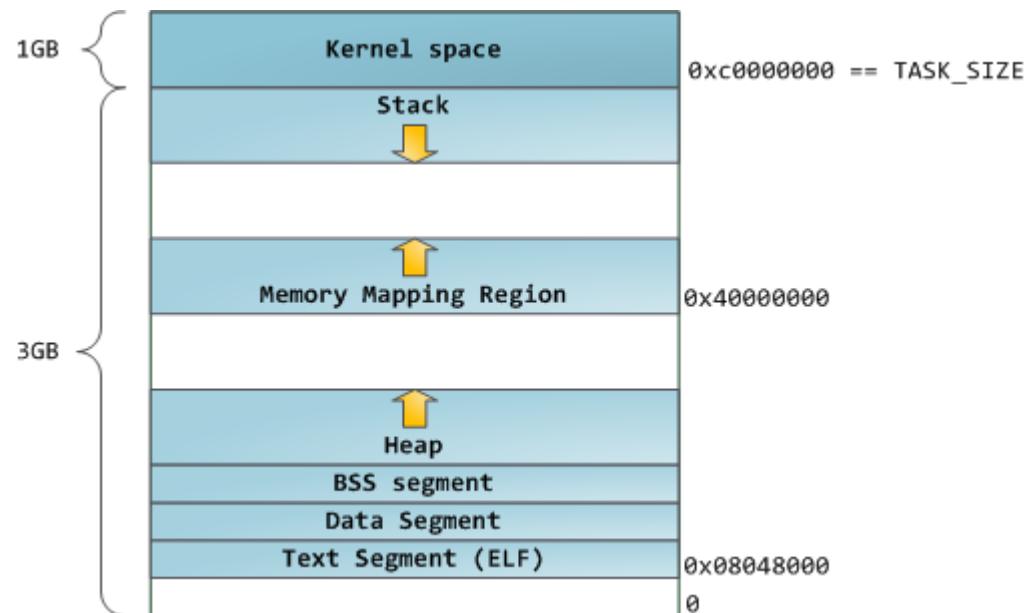
Programmers access it usually through `malloc()`, `realloc()`, `calloc()`, `free()`

Read-write



Large Heap Objects

Allocators may decide to create an anonymous, private mapping to store large objects instead of directly storing into the heap



Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

Intel x86 Processors

Dominate laptop/desktop/server market

Evolutionary design

- Backwards compatible up until 8086, introduced in 1978
- Added more features as time goes on

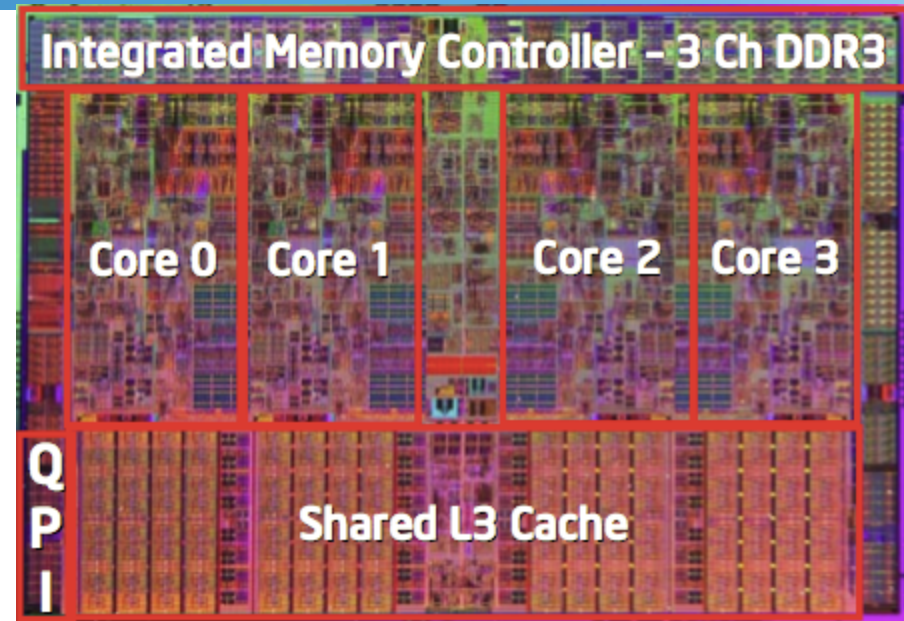
Complex instruction set computer (CISC)

- Many different instructions with many different formats
 - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!
 - In terms of speed. Less so for low power.

Intel x86 Processors

Machine Evolution

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M



Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

x86 Integer Registers

General purpose registers

- On 32-bit architectures
EAX, EBX, ECX, EDX, EDI,
ESI, ESP, EBP

The instruction pointer (IP)

- Also referred to as
program counter (PC)
- EIP on 32-bit

FLAGS register

- Used for control flow
operations, etc.
- EFLAGS

register encoding		high 8-bit	low 8-bit	16-bit	32-bit
0		AH (4)	AL	AX	EAX
3		BH (7)	BL	BX	EBX
1		CH (5)	CL	CX	ECX
2		DH (6)	DL	DX	EDX
6		SI		SI	ESI
7		DI		DI	EDI
5		BP		BP	EBP
4		SP		SP	ESP
		31	16 15	0	
		FLAGS		FLAGS	EFLAGS
		IP		IP	EIP
		31	0		

x86-64 Integer Registers

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

x86-64 Integer Registers

Can reference low-order bytes too

- d suffix for lower 32-bits (r8d)
- w suffix for lower 16-bits (r8w)
- b suffix for lower 8-bits (r8b)

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

Typical Register Uses

EAX: accumulator

EBX : Pointer to data

ECX: Counter for string operations and loops

EDX: I/O Operations

EDI: Destination for string operations

ESP: Stack pointer

EBP: Frame pointer

register encoding		high 8-bit	low 8-bit	16-bit	32-bit
0		AH (4)	AL	AX	EAX
3		BH (7)	BL	BX	EBX
1		CH (5)	CL	CX	ECX
2		DH (6)	DL	DX	EDX
6		SI		SI	ESI
7		DI		DI	EDI
5		BP		BP	EBP
4		SP		SP	ESP
	31	16	15	0	

	FLAGS	FLAGS	EFLAGS
	IP	IP	EIP
	31	0	

Assembly Syntax

Intel: OP dest, src

AT&T: OP src, dest

Unix systems prefer AT&T

- We are going to use the same as the GNU assembler (gas syntax)

Assembly Instructions

pushq: push quad word to stack

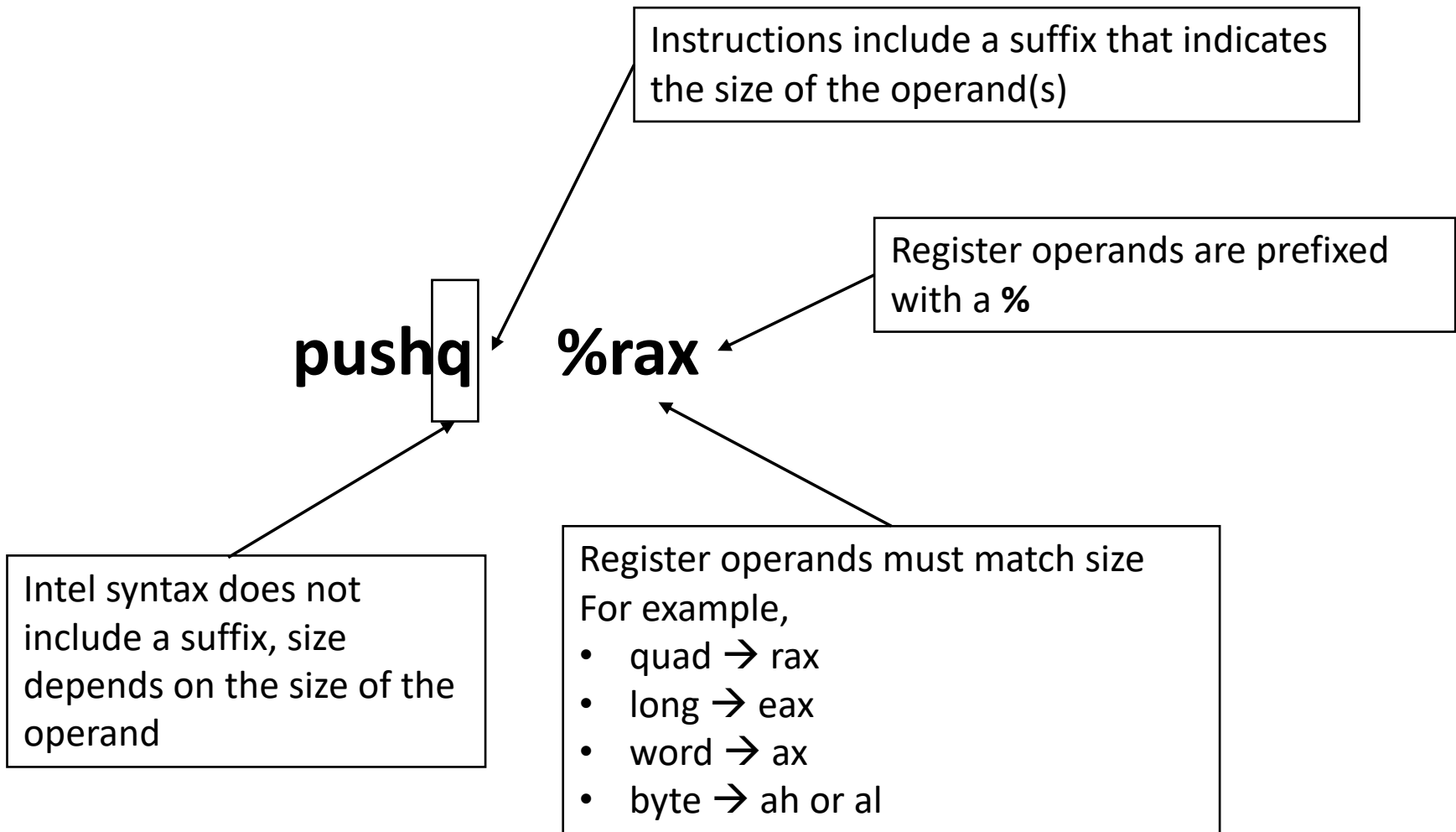
movq: Move quad word

imull: Signed multiply long

addl: Add long

```
pushq  %rbp
movq   %rsp, %rbp
movl   %edi, -20(%rbp)
movl   %esi, -24(%rbp)
movl   %edx, -28(%rbp)
movl   -20(%rbp), %eax
imull  -28(%rbp), %eax
movl   %eax, %edx
movl   -24(%rbp), %eax
addl   %edx, %eax
imull  -28(%rbp), %eax
```

Operand Sizes



Memory Operands

Parentheses indicate a memory operand

Each memory address can be defined as:

Base+Index*Scale+Disp

- In AT&T syntax:
 disp(base, index, scale)
- disp, index, and scale are optional

```
pushq %rbp
movq  %rsp, %rbp
movl  %edi, -20(%rbp)
movl  %esi, -24(%rbp)
movl  %edx, -28(%rbp)
movl  -20(%rbp), %eax
imull -28(%rbp), %eax
movl  %eax, %edx
movl  -24(%rbp), %eax
addl  %edx, %eax
imull -28(%rbp), %eax
```

Memory Addressing Modes

Normal (B) Mem[Reg[R]]

- Register R specifies memory base address
- Pointer dereferencing in C

```
movq (%rcx) , %rax
```

Displacement D(B) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp) , %rdx
```

Memory Addressing Modes

Most General Form

$$D(B,I,S) \qquad \text{Mem}[\text{Reg}[R_b] + S * \text{Reg}[R_i] + D]$$

- D: Constant “displacement” 1, 2, or 4 bytes
- R_b: Base register
- R_i: Index register: Any, except for **%rsp**
- S: Scale: 1, 2, 4, or 8

```
movq 8(%rbp, %rax, 4), %rdx
```

Immediates

Constants or immediates are defined using \$

addl \$1, %eax

In decimal, unless:

- 0x prefix is used → hexadecimal
- 0 prefix is used → octal

Immediates can help you identify the syntax

Endianness

Memory representation of multi-byte integers

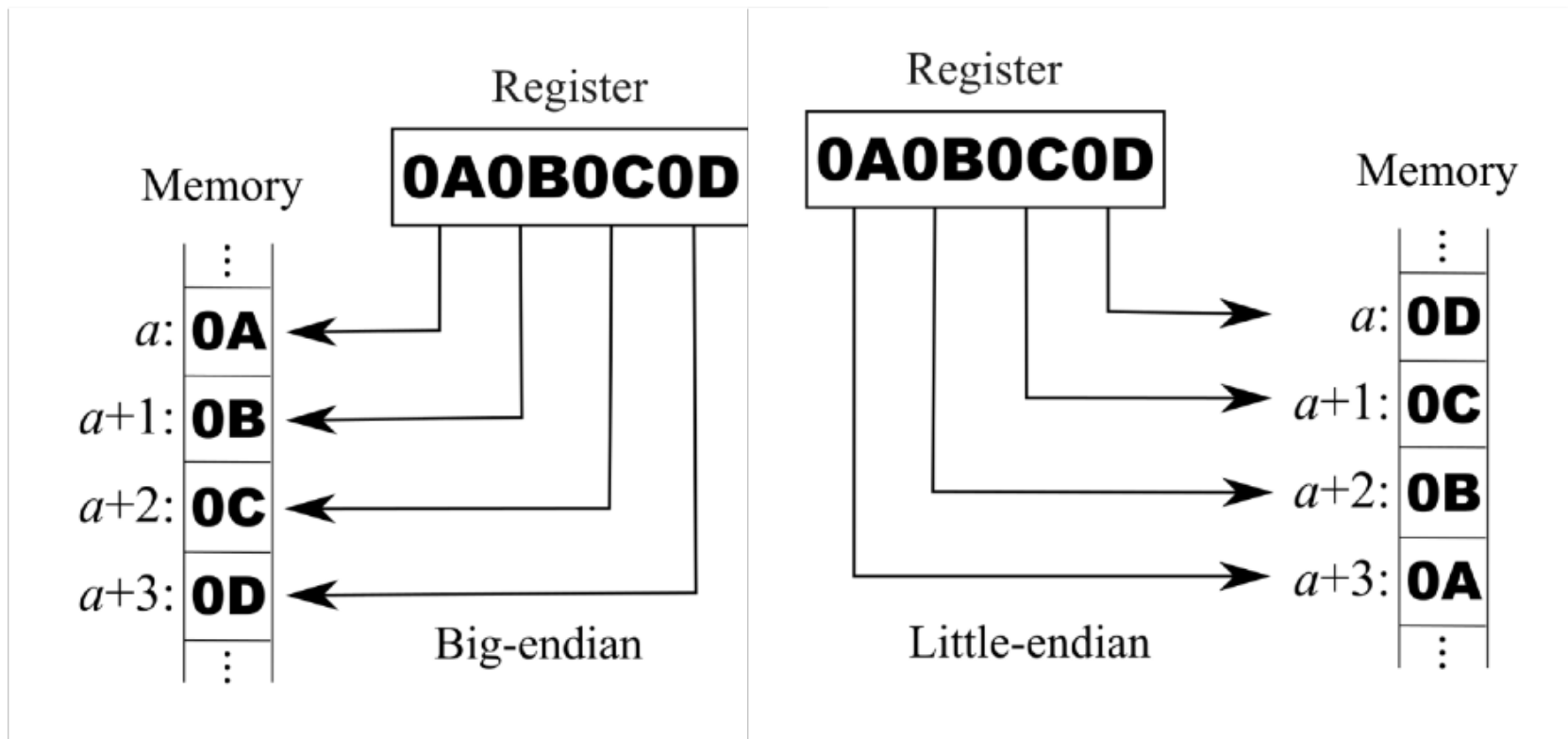
For example the integer: 0x0A0B0C0Dh

Big-endian $\longleftrightarrow$ highest order byte first

0A 0B 0C 0D

Little-endian $\longleftrightarrow$ lowest order byte first (X86)

0D 0C 0B 0A



Load Effective Address

`leaq Src, Dst`

- Src is address mode expression
- Set Dst to address denoted by expression

Computing addresses without a memory reference

- E.g., translation of `p = &x[i];`

Computing arithmetic expressions of the form $x + k * y$

- $k = 1, 2, 4, \text{ or } 8$

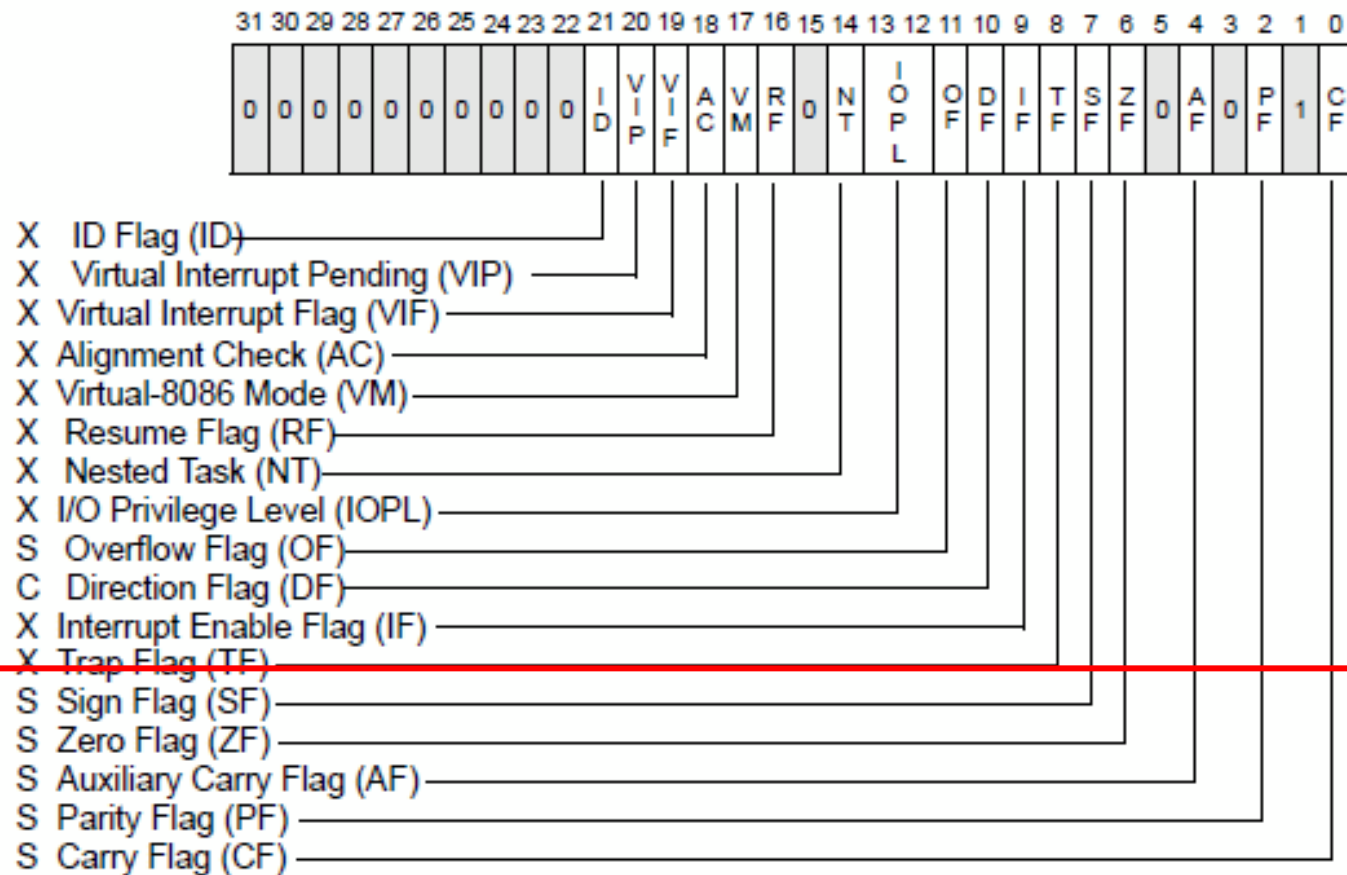
Example

```
leaq (%rdi,%rdi,2), %rax
```

Control Flow

```
if (a > b) {  
    c = d;  
} else {  
    d = c;  
}
```

```
13:  cmp    -0x8(%rbp), %eax  
16:  jle     0xe  
18:  mov     -0x10(%rbp), %eax  
1b:  mov     %eax, -0xc(%rbp)  
1e:  jmp     0x6  
20:  mov     -0xc(%rbp), %eax  
23:  mov     %eax, -0x10(%rbp)  
26:  mov     -0xc(%rbp), %eax
```



S Indicates a Status Flag
 C Indicates a Control Flag
 X Indicates a System Flag

Reserved bit positions. DO NOT USE.
 Always set to values previously read.

EFLAGS Register

Condition Codes Set by Compare

Explicit Setting by Compare Instruction

- `cmpq Src2, Src1`
- `cmpq b, a` like computing $a - b$ without setting destination
- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if $a == b$
- **SF set** if $(a - b) < 0$ (as signed)
- **OF set** if two's-complement (signed) overflow
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (a - b) < 0) \ || \ (a < 0 \ \&\& \ b > 0 \ \&\& \ (a - b) > 0)$

Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

- `testq Src2, Src1`
 - `testq b, a` like computing `a&b` without setting destination
- Sets condition codes based on value of Src1 & Src2
- Useful to have one of the operands be a mask
- **ZF set** when `a&b == 0`
- **SF set** when `a&b < 0`

Common Conditional Jumps

jX	Condition	Description
jmp	1	Unconditional
j _e	ZF	Equal / Zero
j _{ne}	$\sim ZF$	Not Equal / Not Zero
j _g	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
j _{ge}	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
j _l	$(SF \wedge OF)$	Less (Signed)
j _{le}	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)

Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

Function Calls

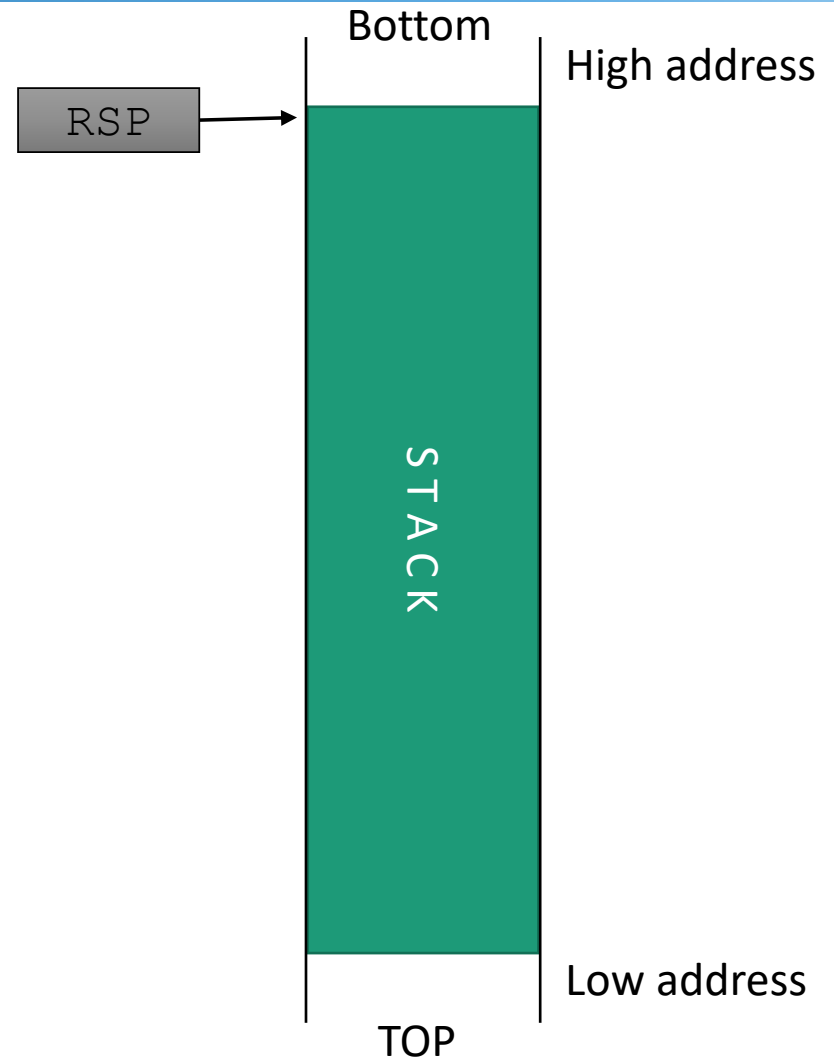
Function calls transfer control and use the stack to keep track of callees

- `call <address>` Transfer control to address and save the address of the next instruction on the stack
- `ret` Pop the address from the stack and transfer control to it

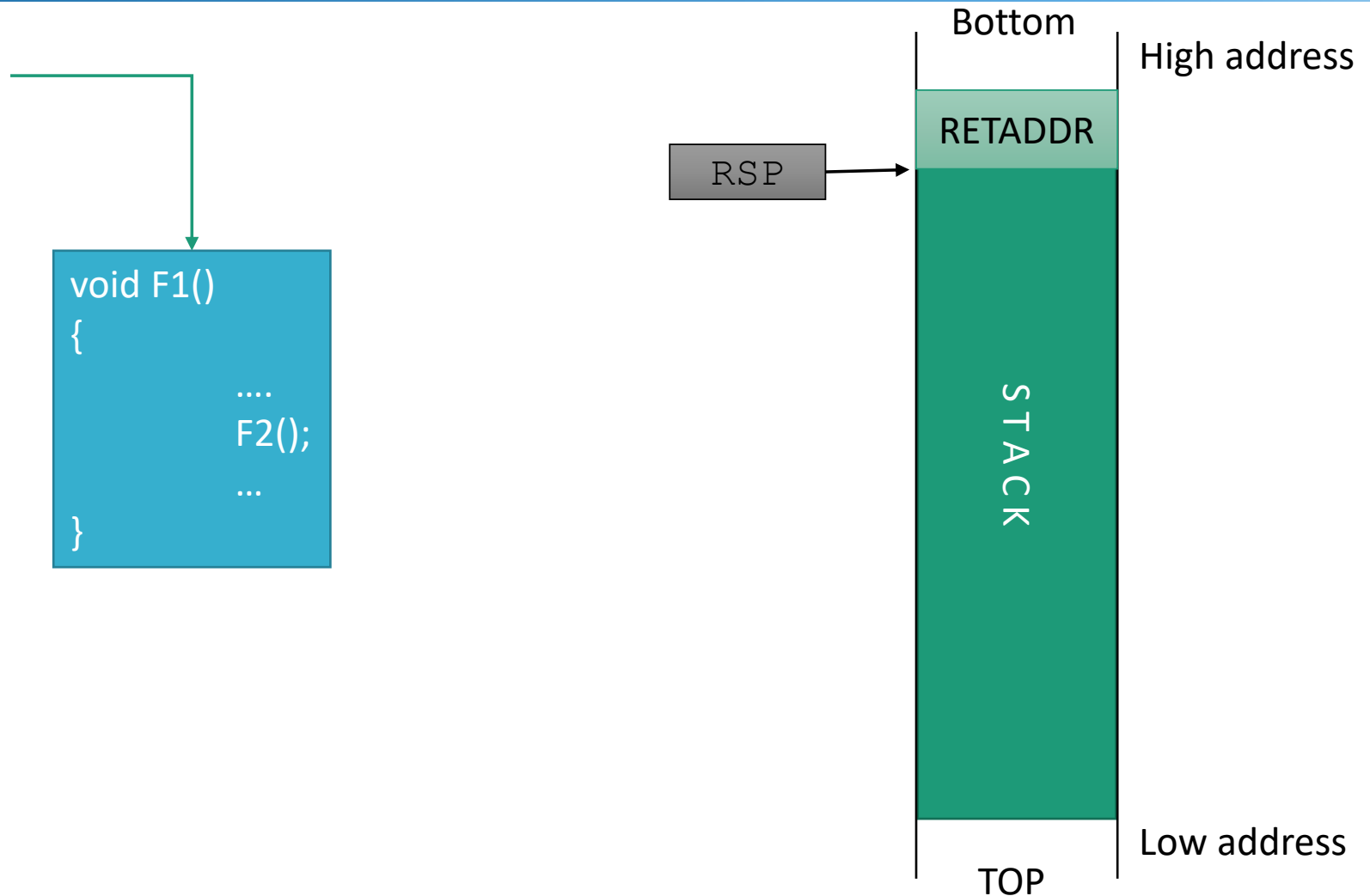
`call` and `ret` implicitly use the RSP register

- So does `push/pop`

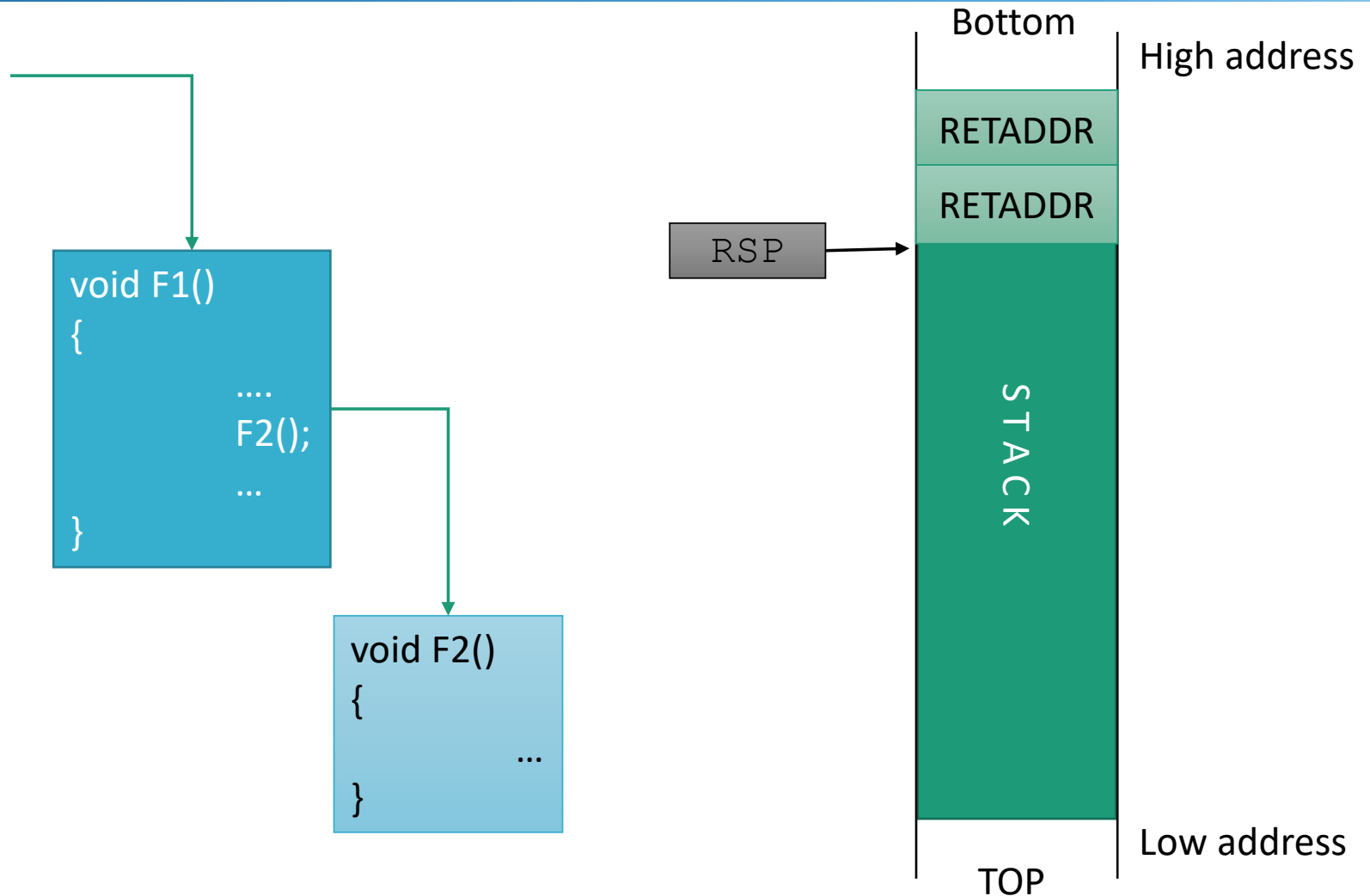
Simple Function Call



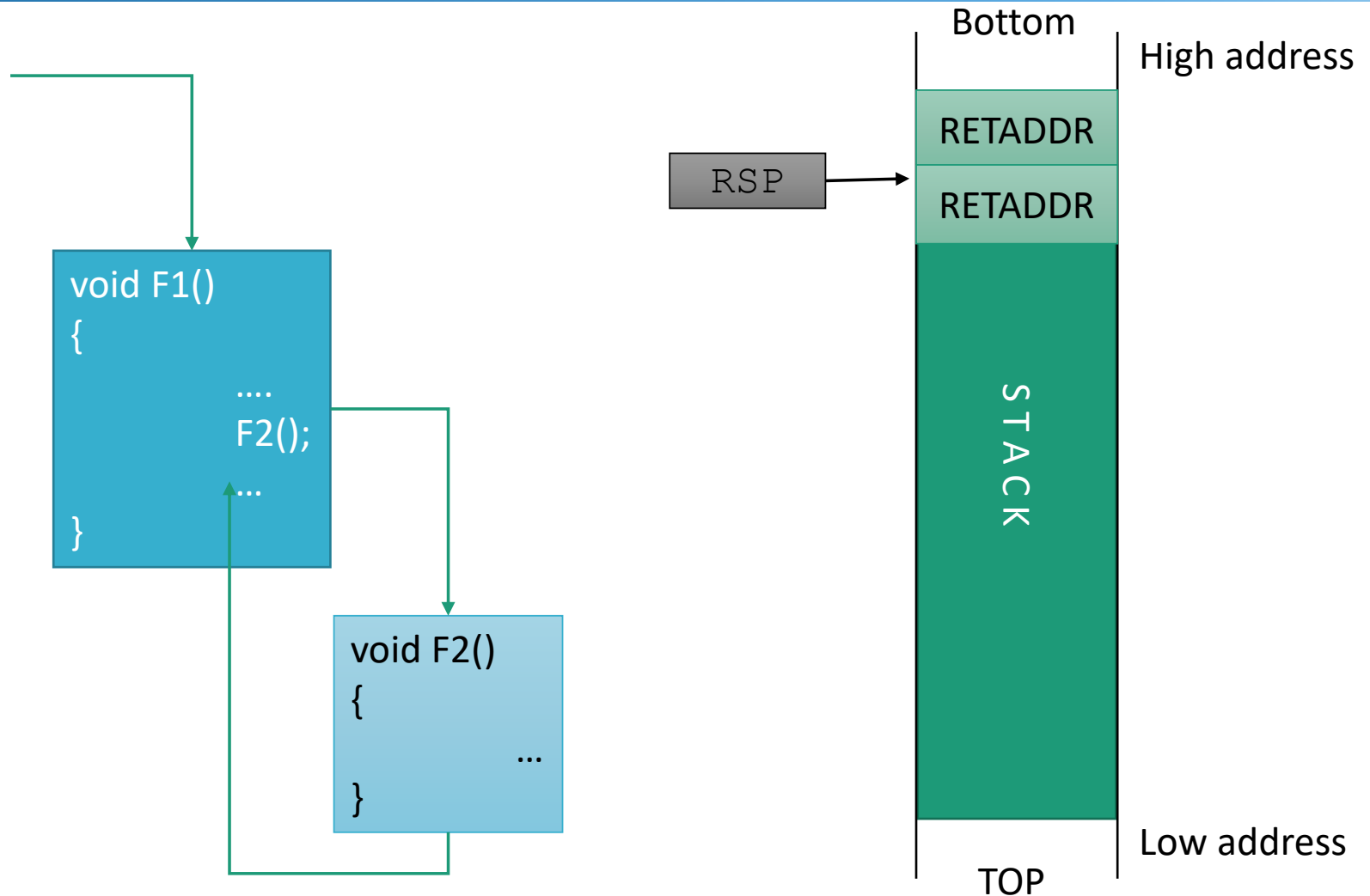
Simple Function Call



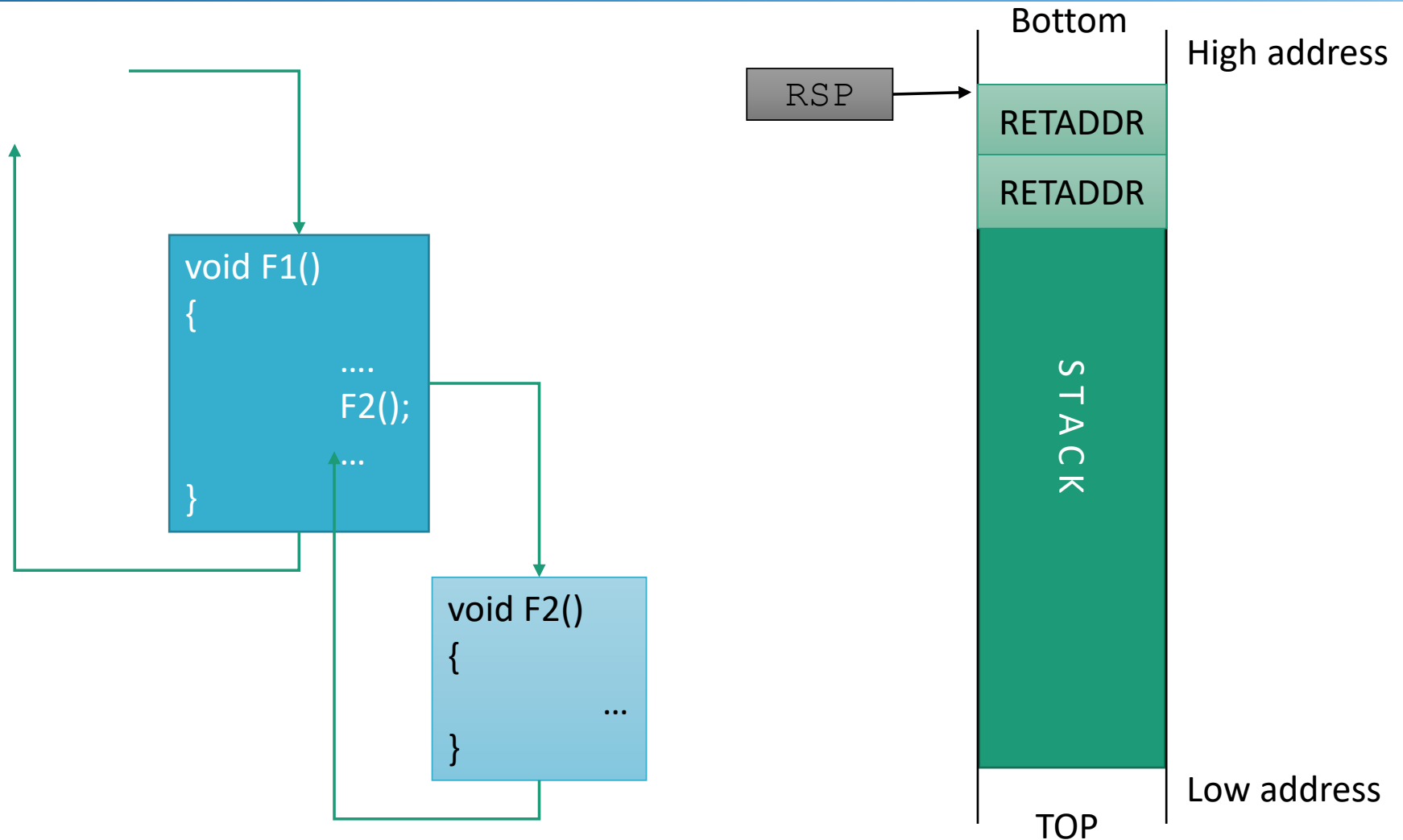
Simple Function Call



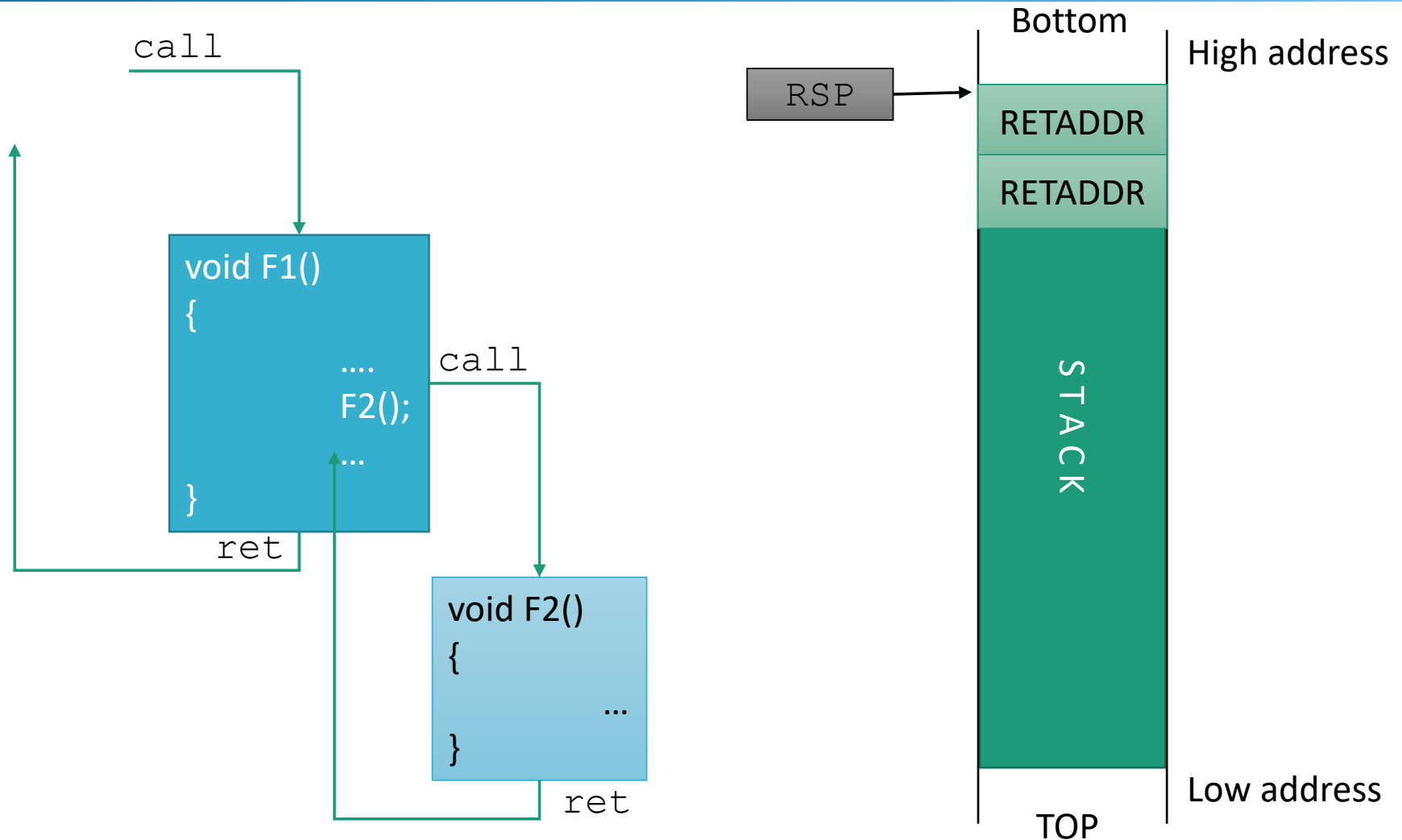
Simple Function Call



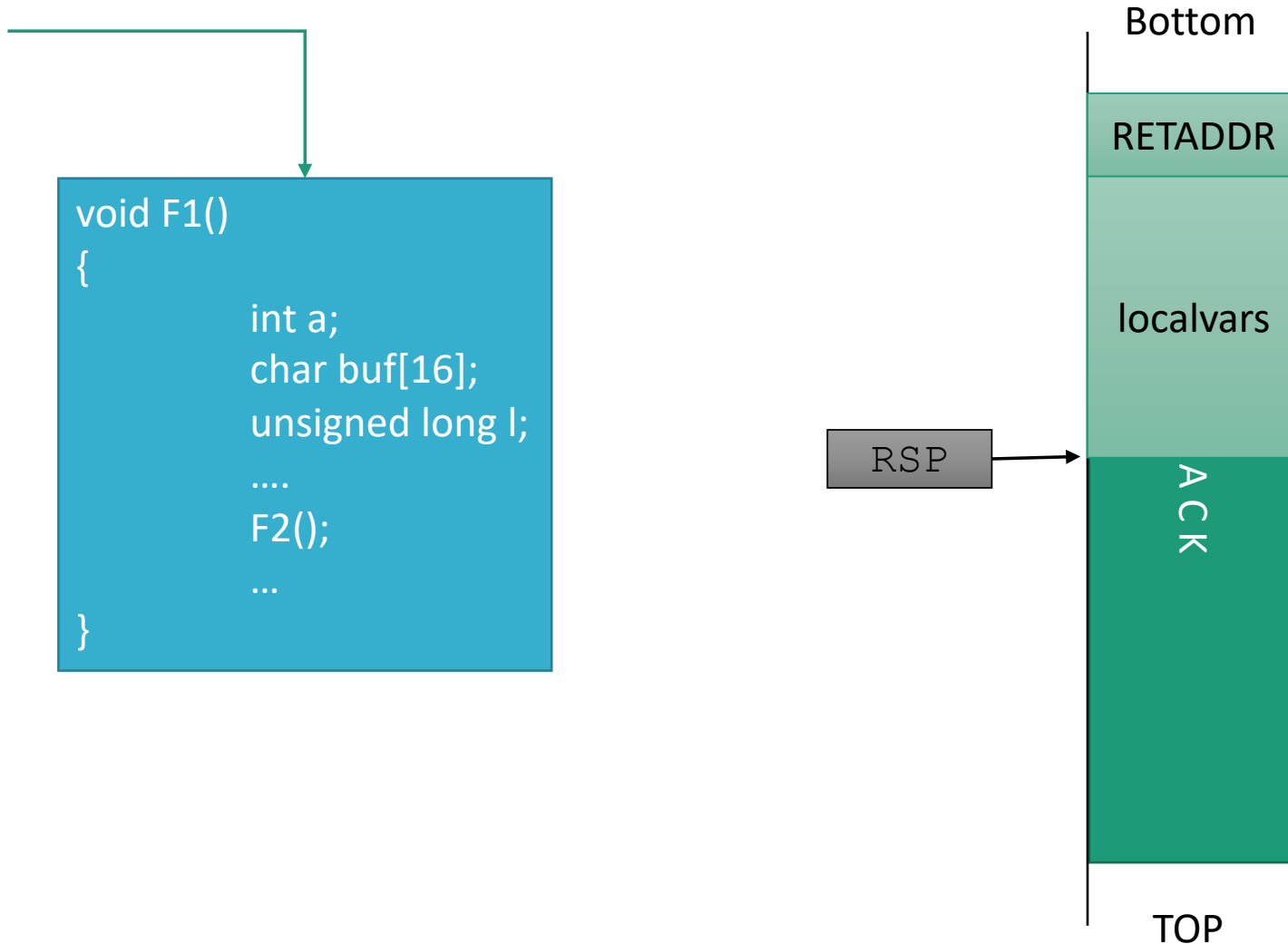
Simple Function Call



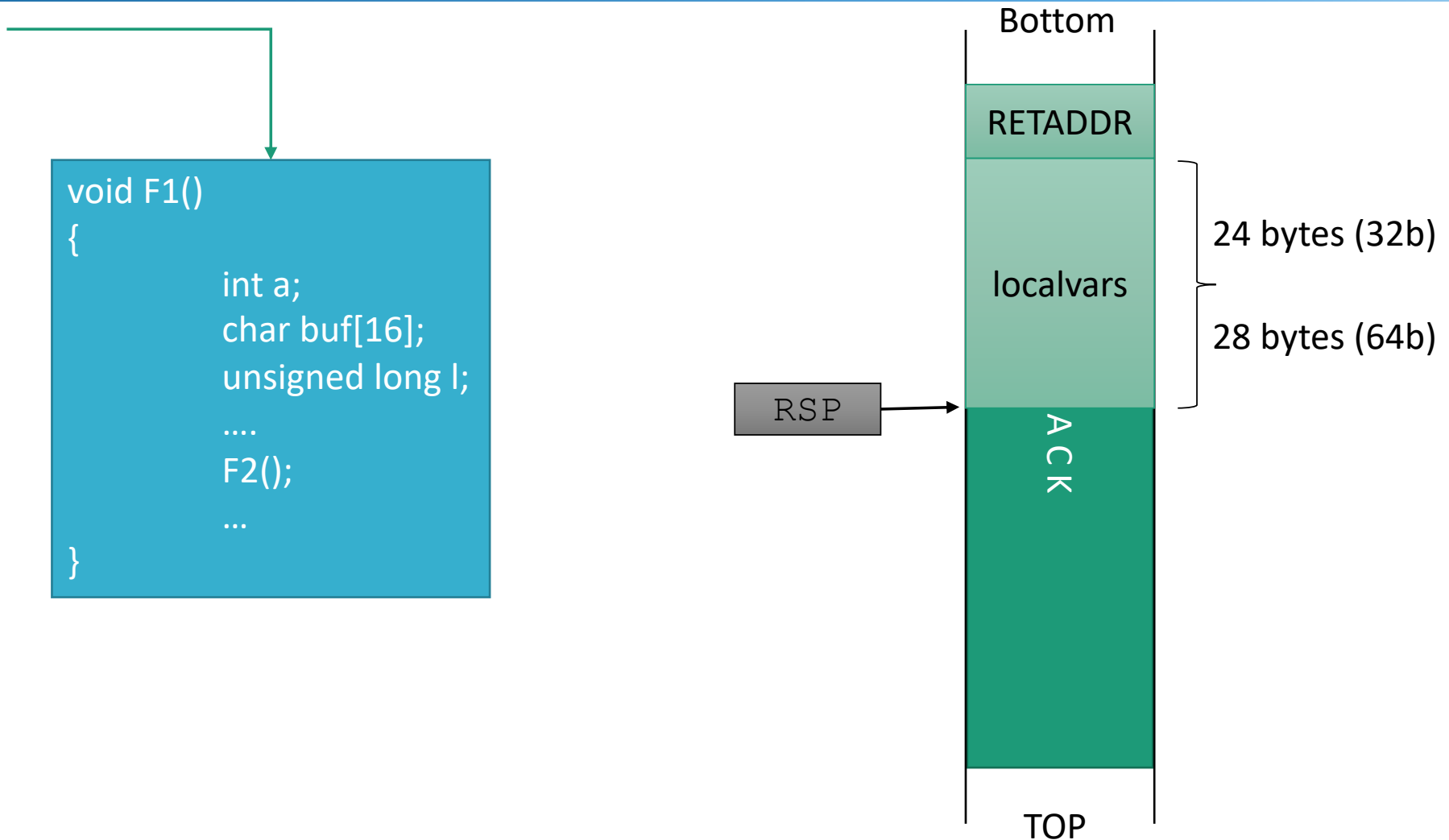
Simple Function Call



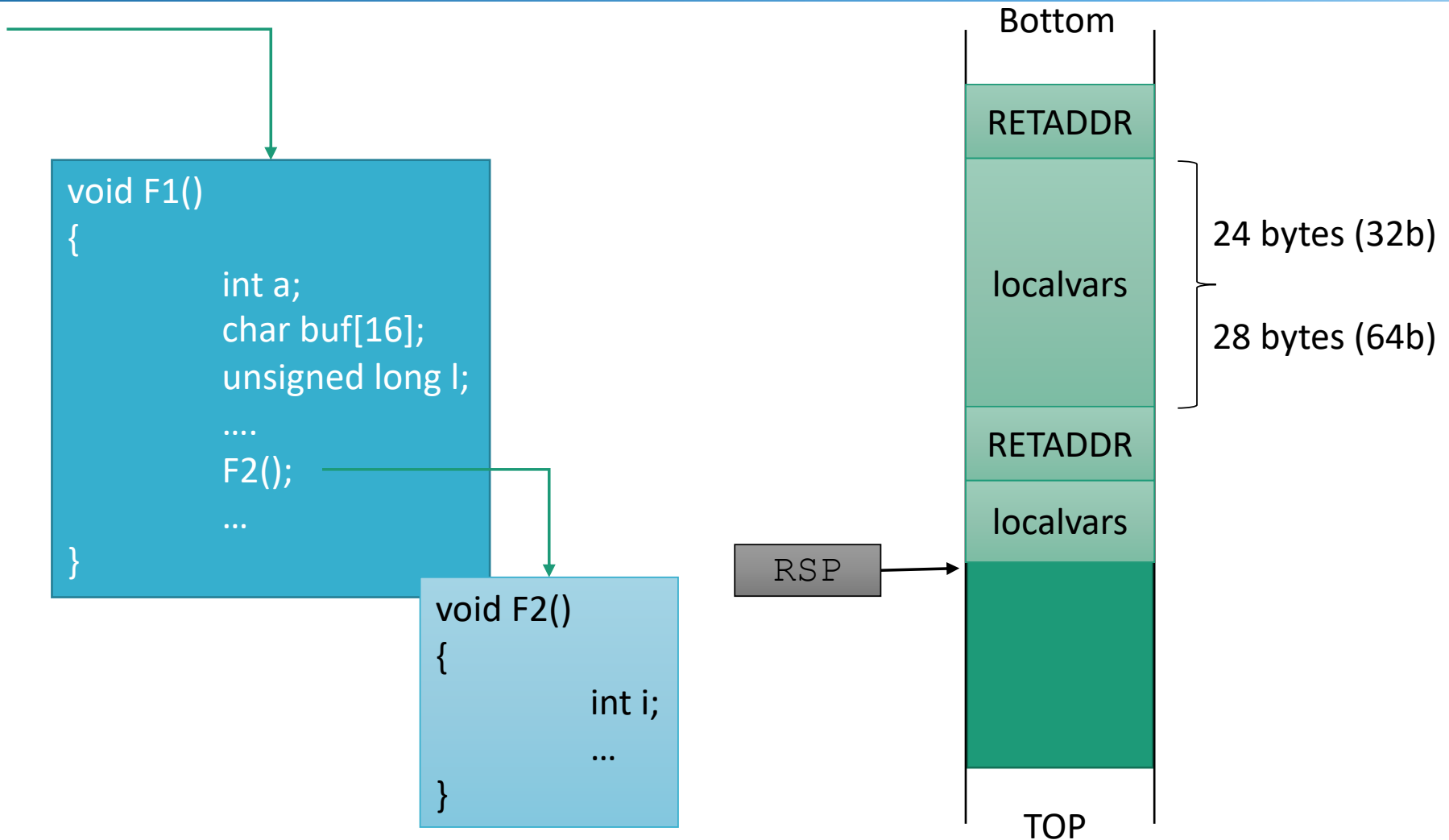
Local Variables



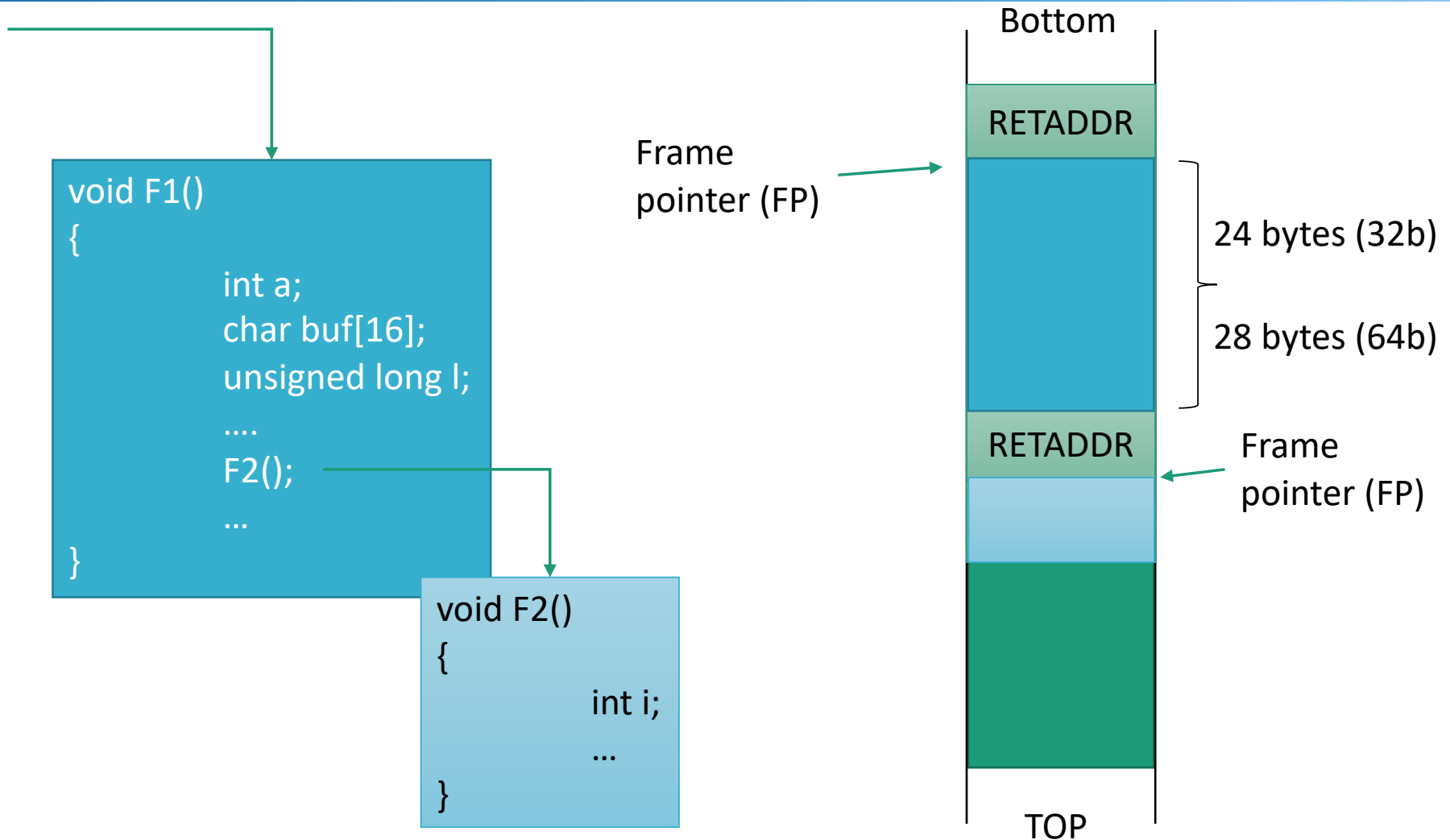
Local Variables



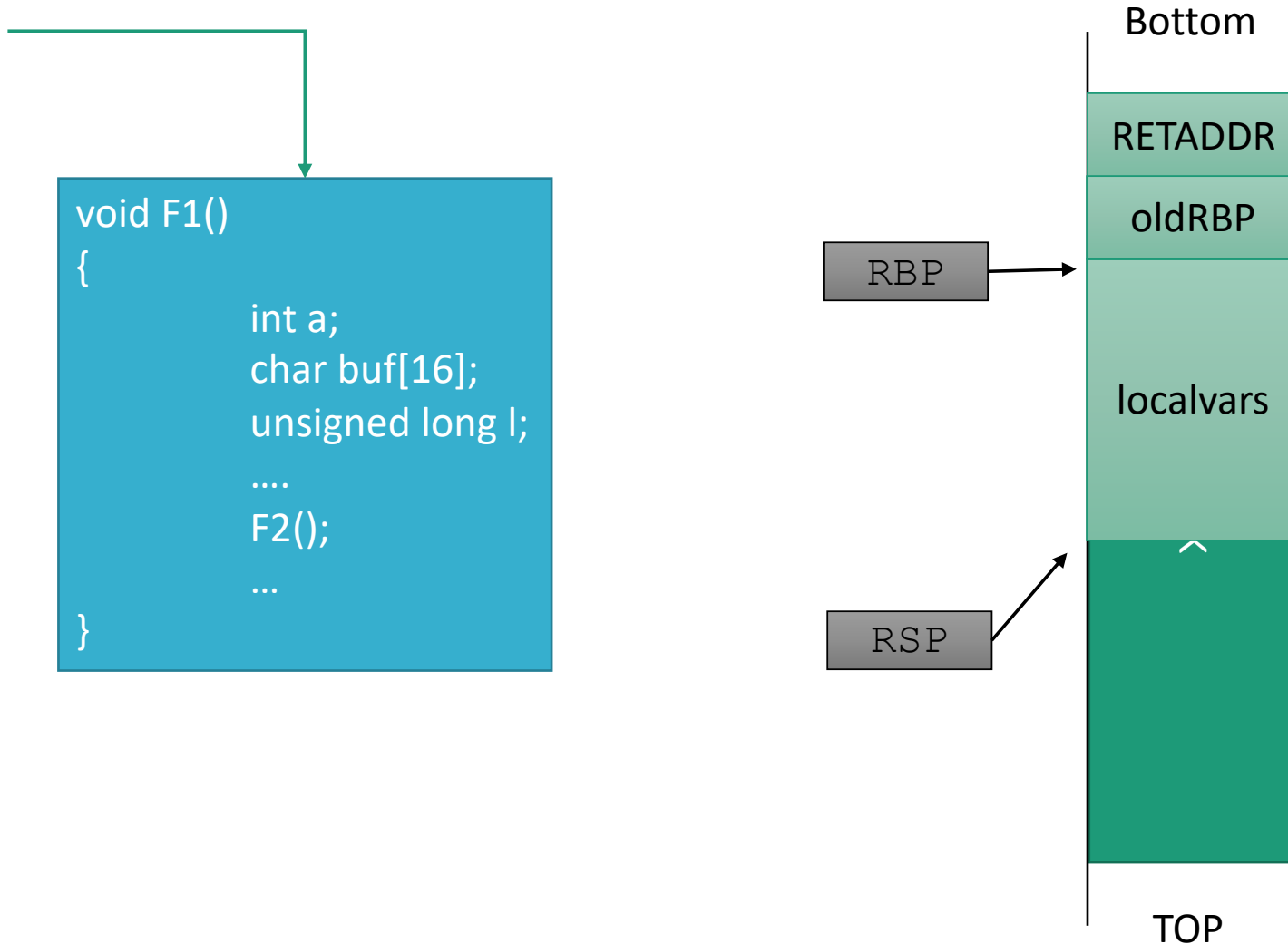
Local Variables



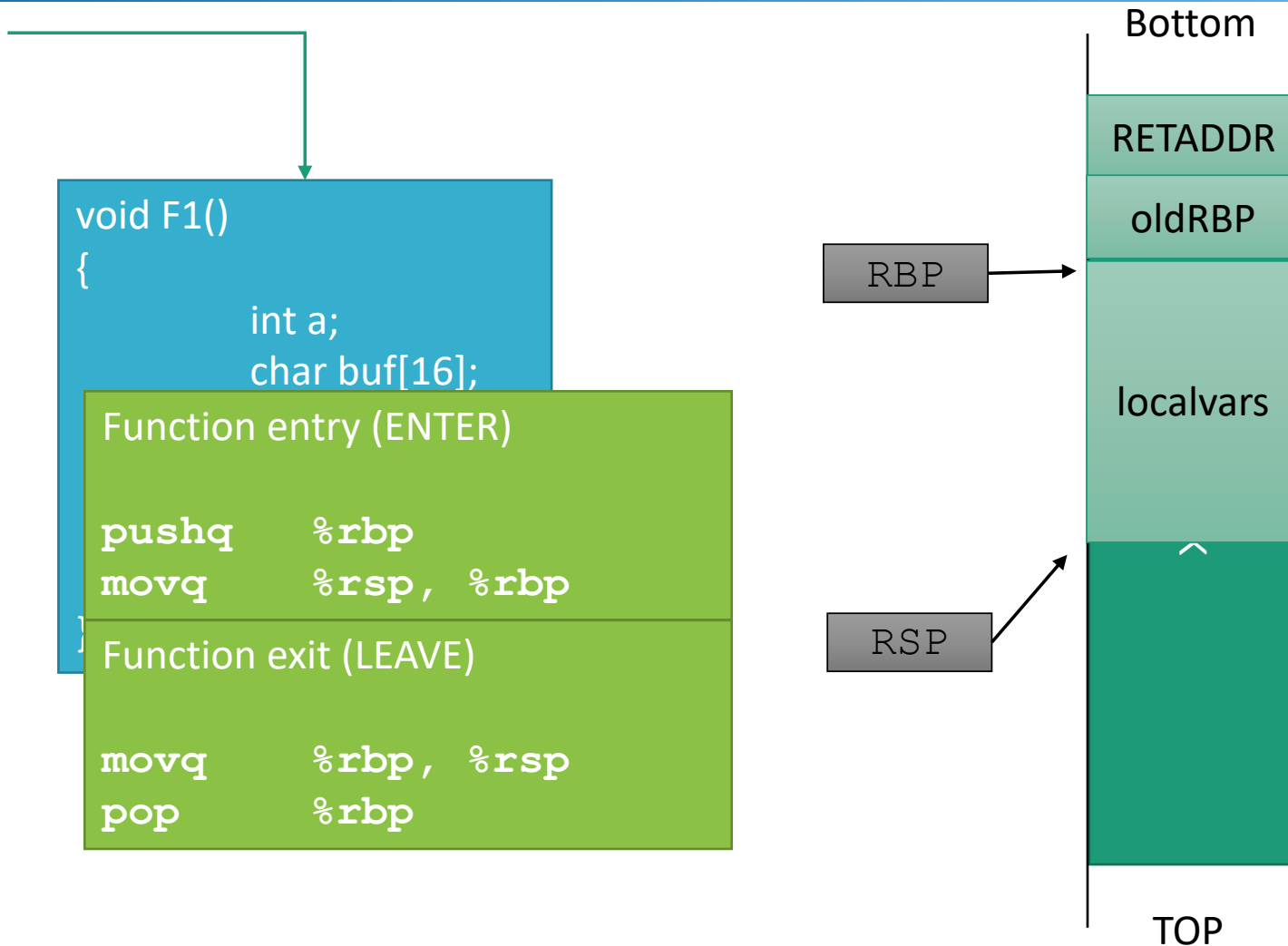
Stack Frames



RBP == Frame Base Pointer



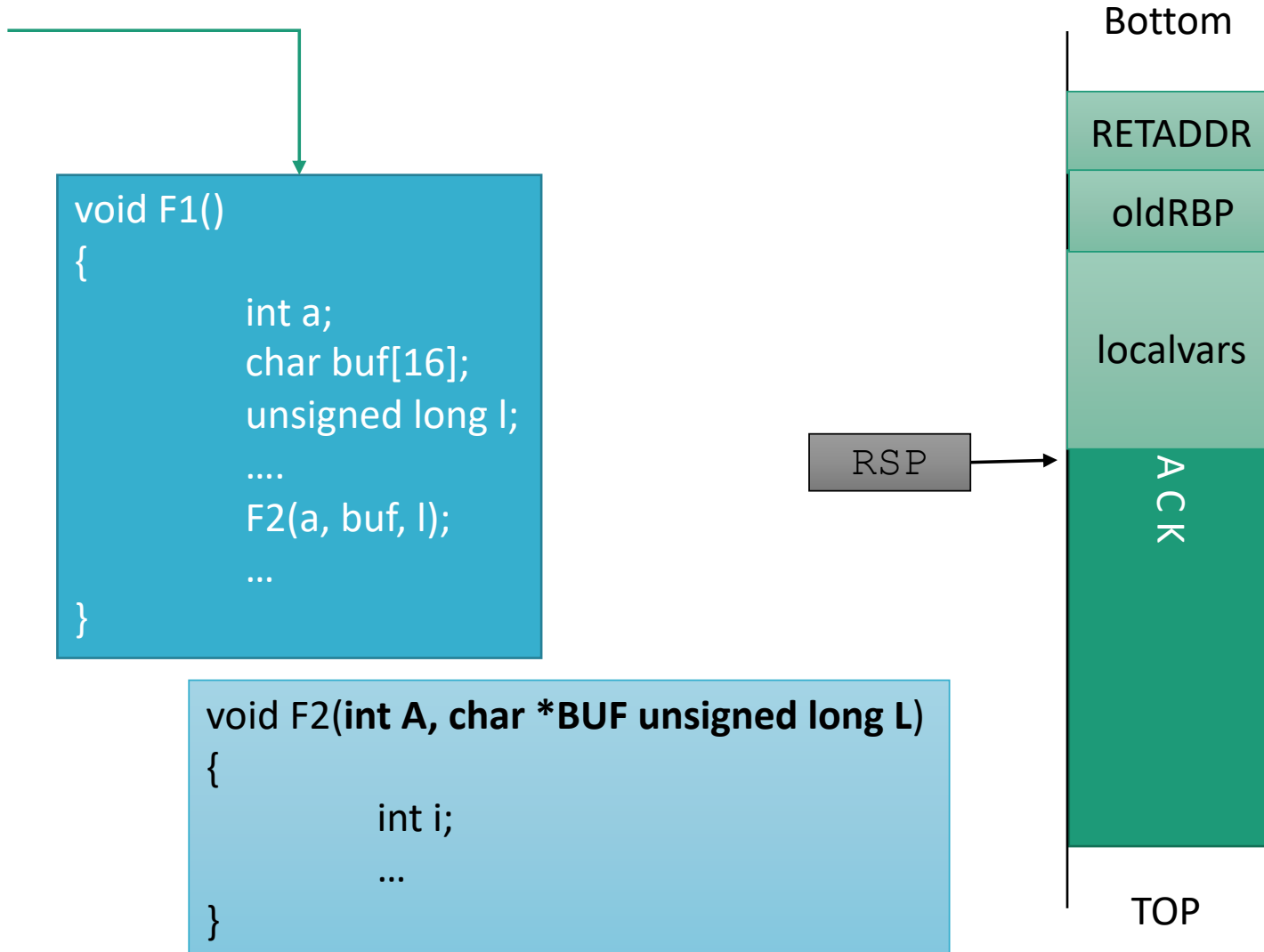
RBP == Frame Base Pointer



Frame pointers are optional

```
gcc -fomit-frame-pointer test.c
```

Function Arguments



Calling Conventions

Defines the standard for passing arguments

Caller and callee need to agree

Enforced by compiler

Important when using 3rd party libraries

Different styles $\leftrightarrow$ different advantages

Popular Conventions

cdecl (mostly 32-bit)

Arguments are passed on the stack

- Pushed right to left

eax, edx, ecx are caller saved

- callee can overwrite without saving

ebx, esi, edi are callee saved

- callee must ensure they have same value on return

eax used for function return value

System V AMD64 ABI

Arguments are passed using registers

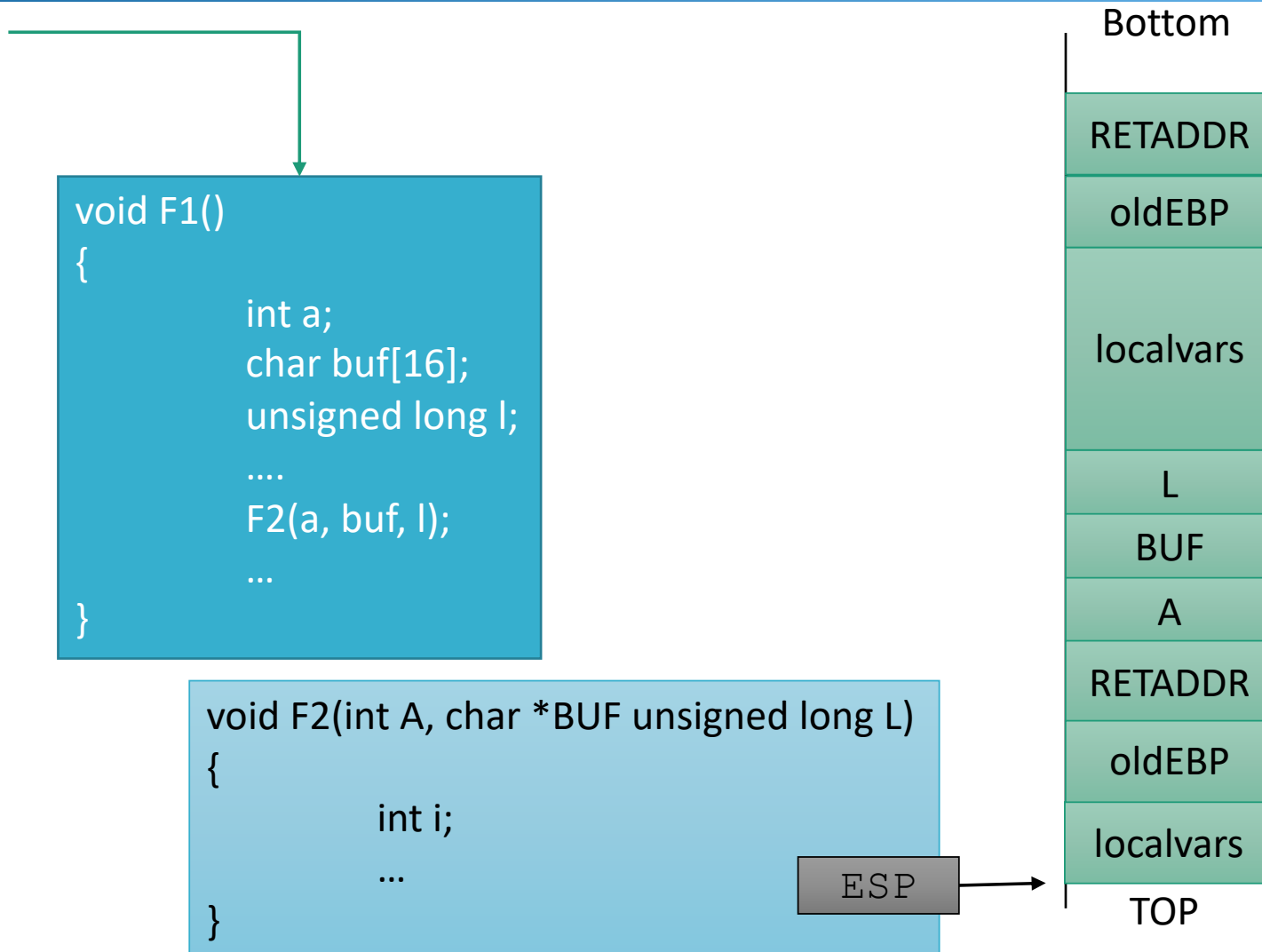
- First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9

RBP, RBX, and R12–R15 are callee saved

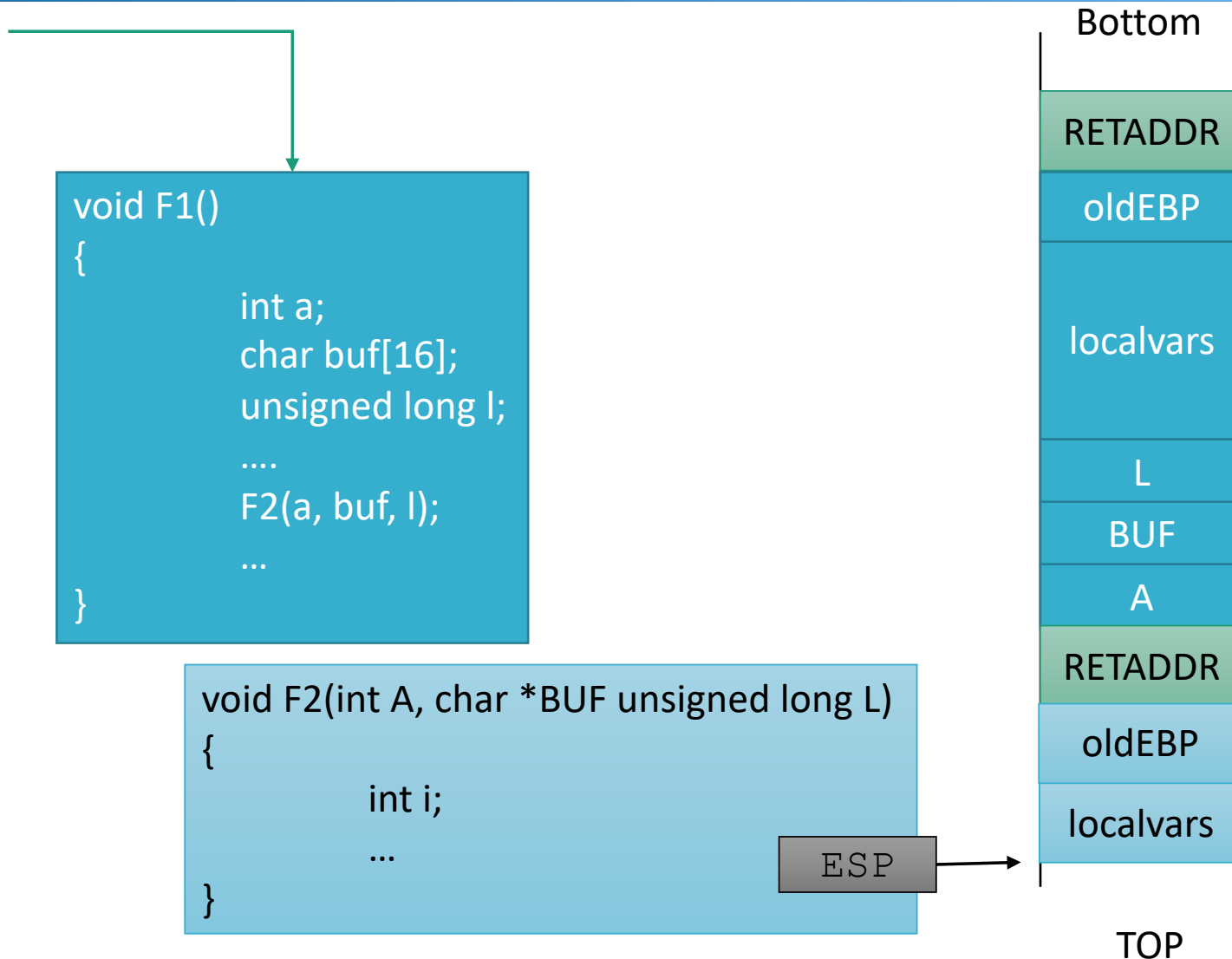
RAX used for function return

https://en.wikipedia.org/wiki/X86_calling_conventions

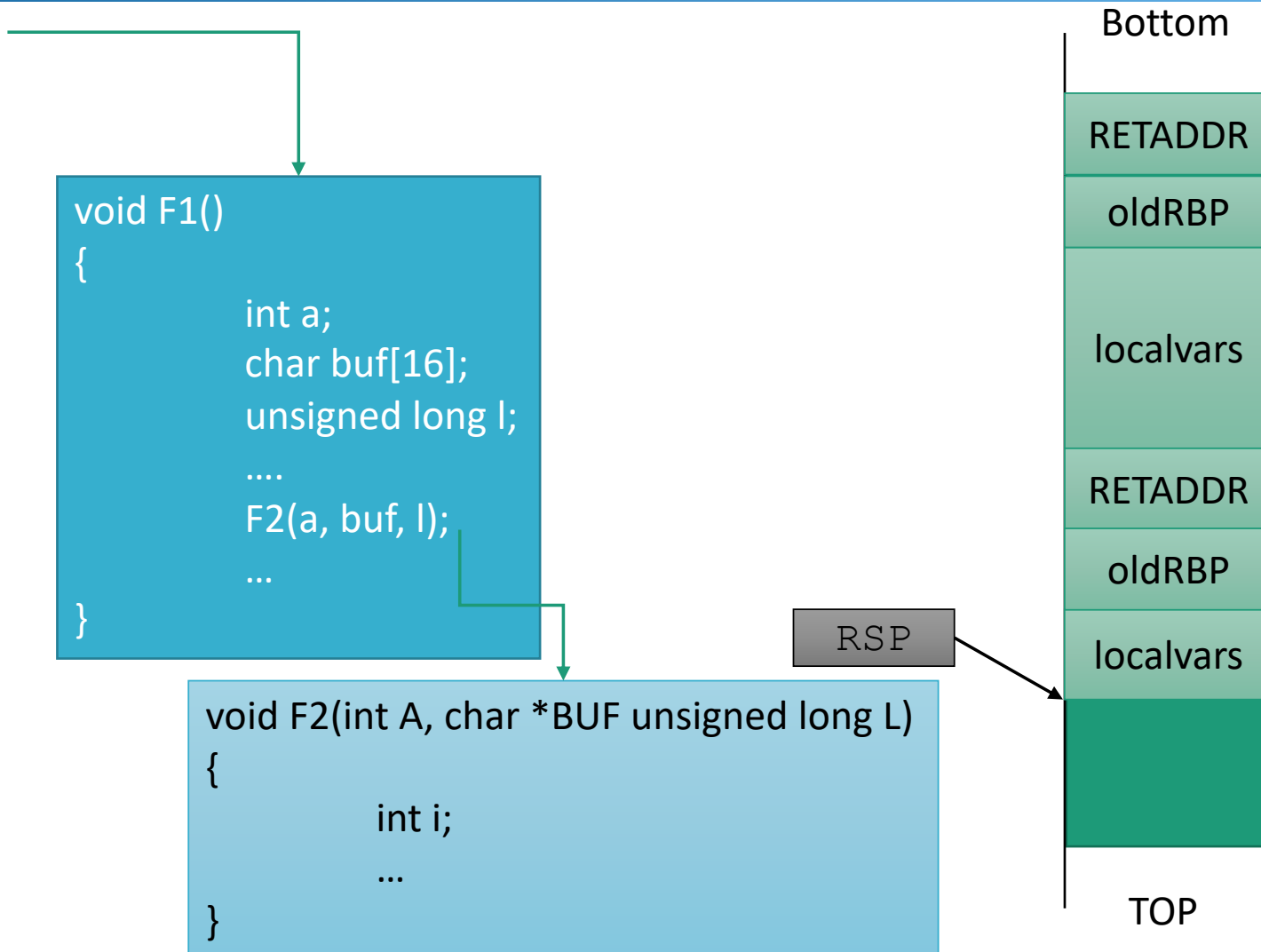
cdecl Example



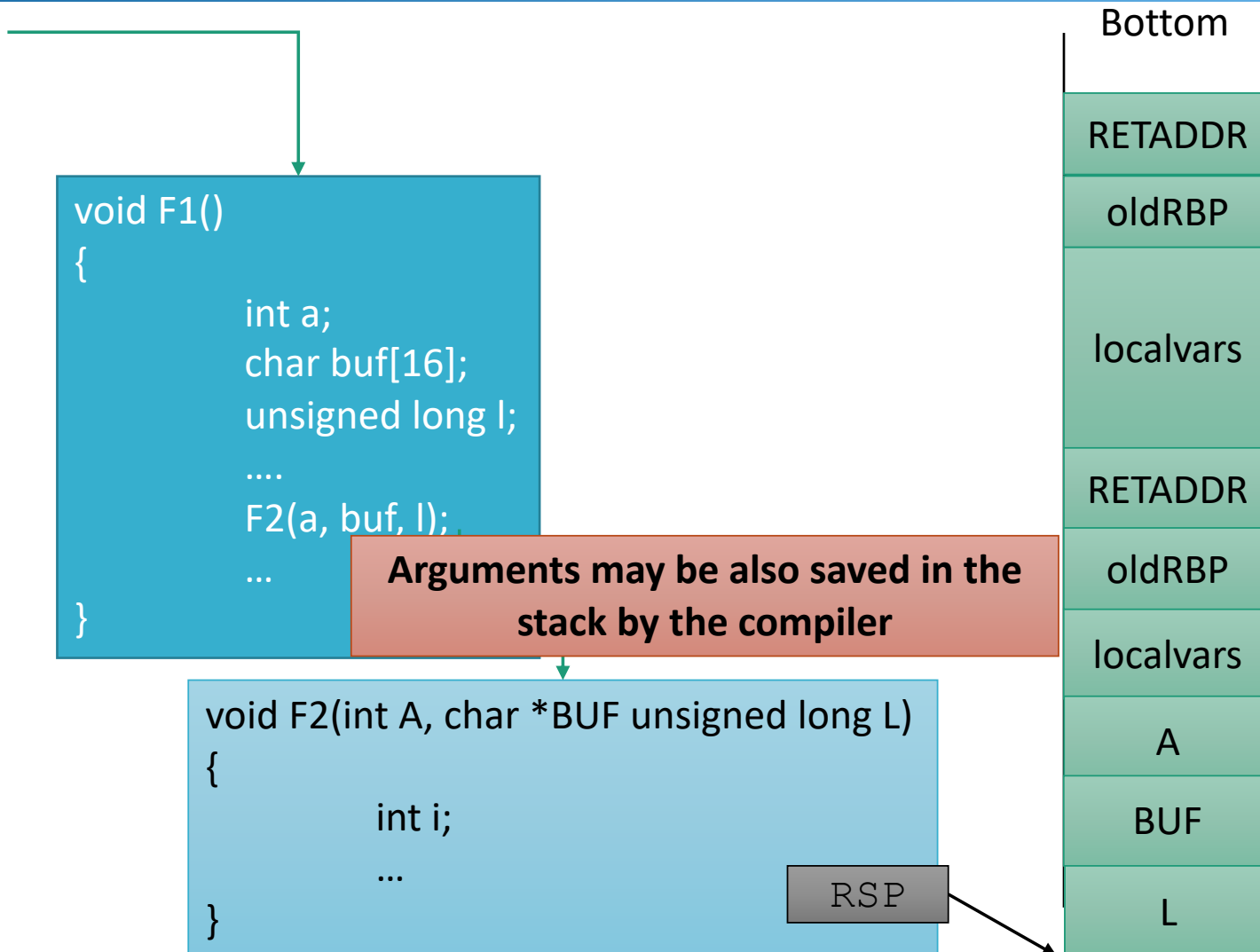
cdecl Example



System V AMD64 ABI Example



Register/Argument Spilling



Variable Number of Arguments

Used in variadic functions, like

```
int printf(const char *format, ...);
```

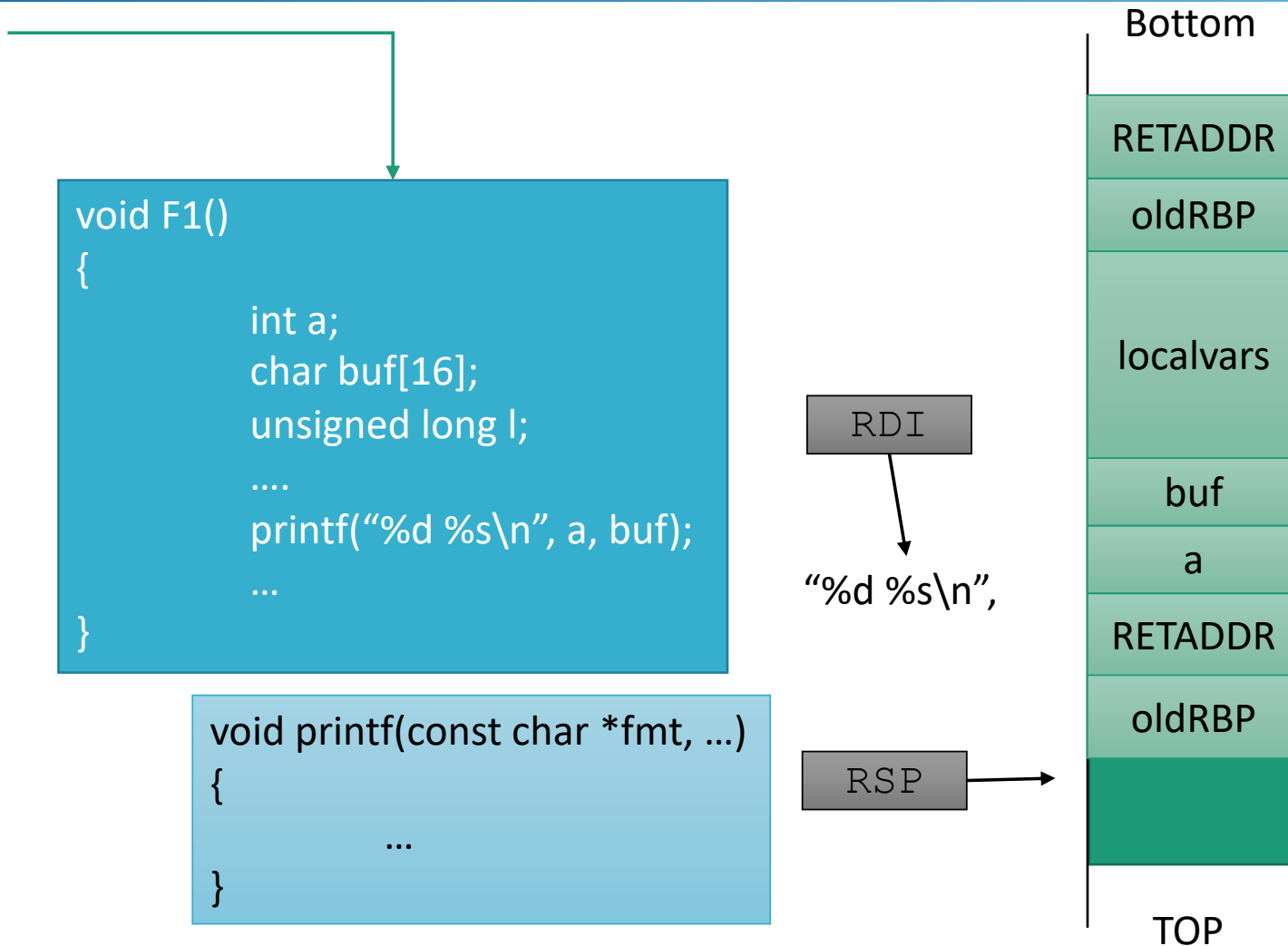
Arguments passed in the stack

- Order right-to-left

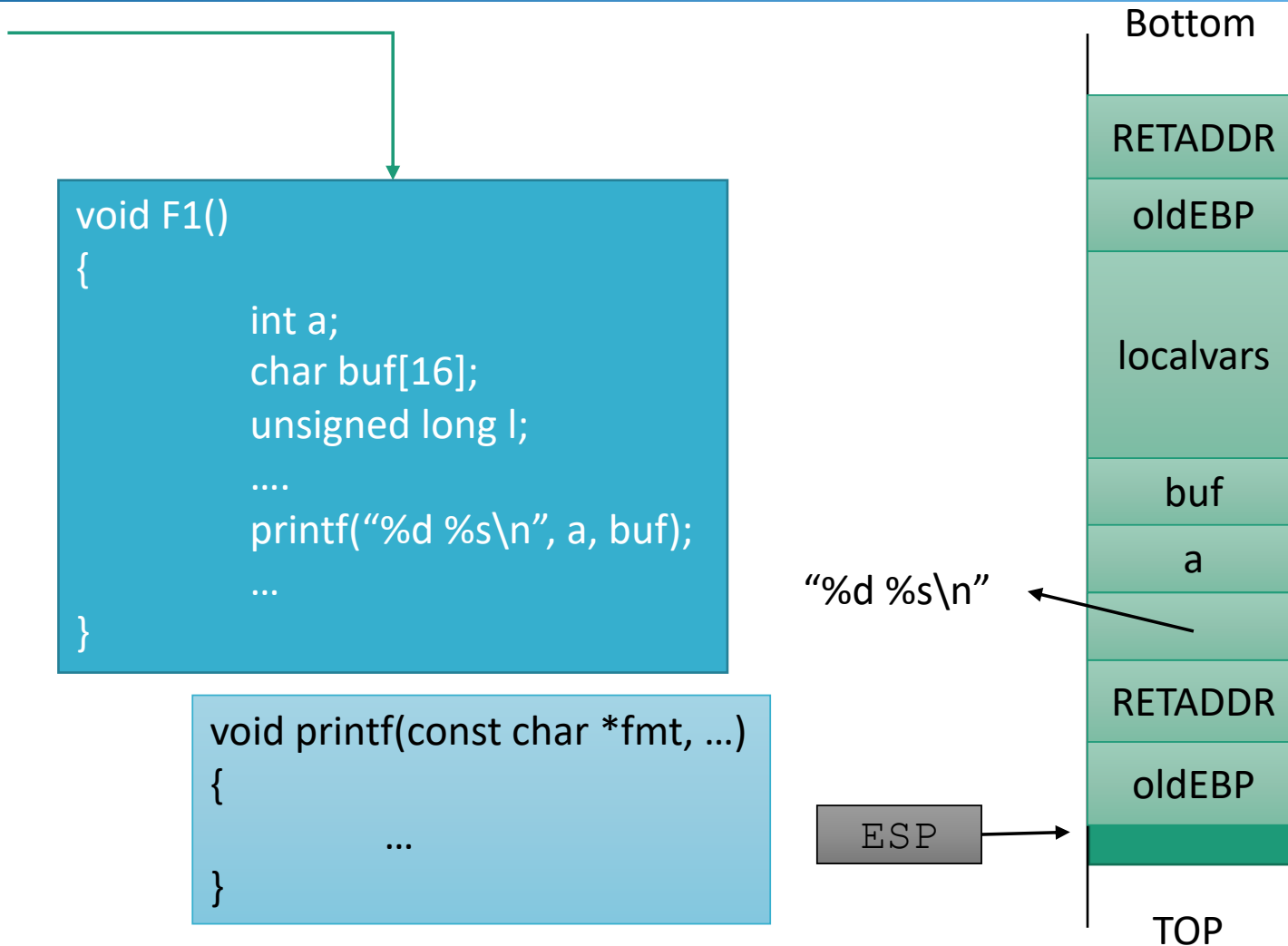
Only caller knows exact number of arguments

- Caller responsible for cleaning

Example on 64-bit



Example on 32-bit

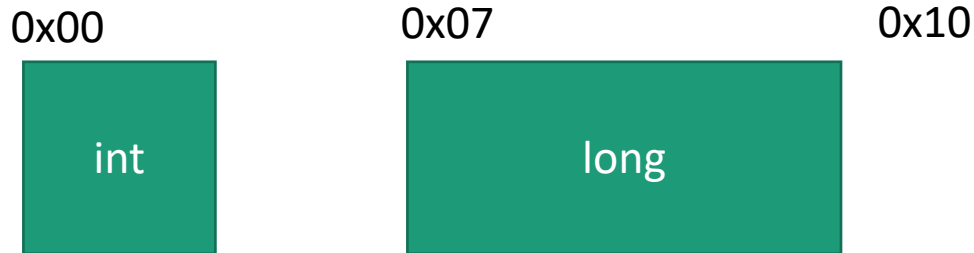


Alignment

CPUs like aligned data

- Better performance

Compilers try to align data



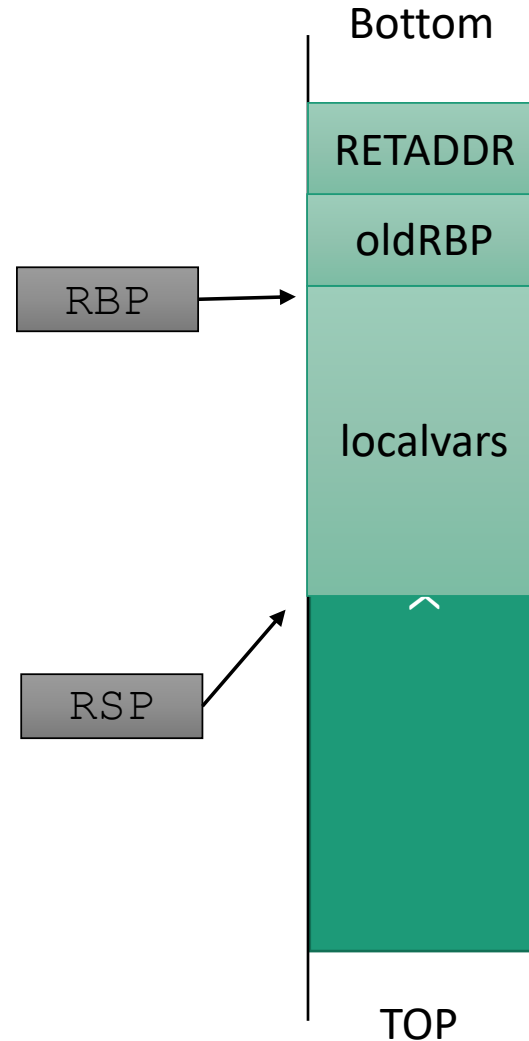
Accessing Stack Variables

With frame pointer

```
mov    -0x18(%rbp), %eax
```

With stack pointer

```
mov    0xc(%rsp), %eax
```



Overview

Introduction

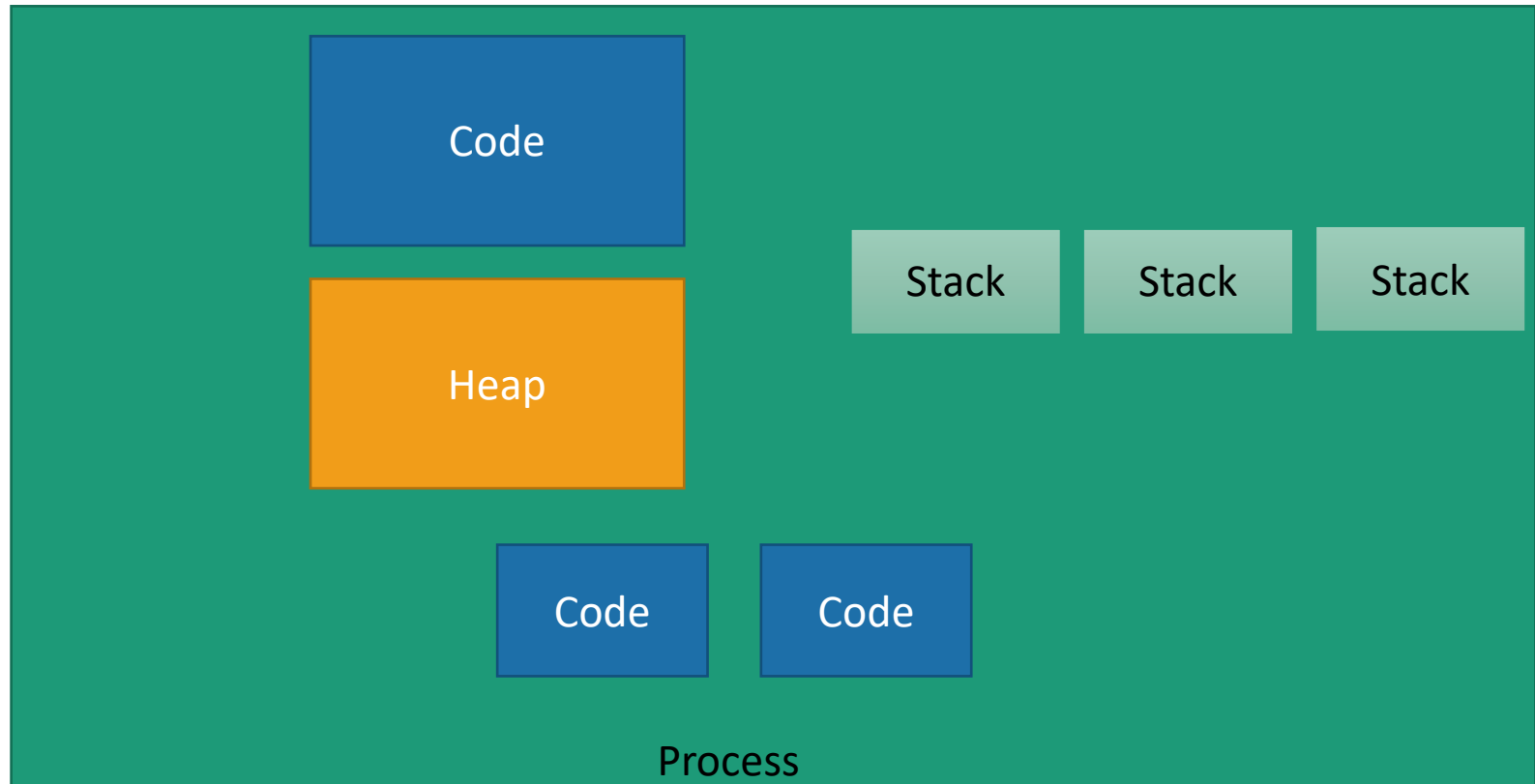
Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

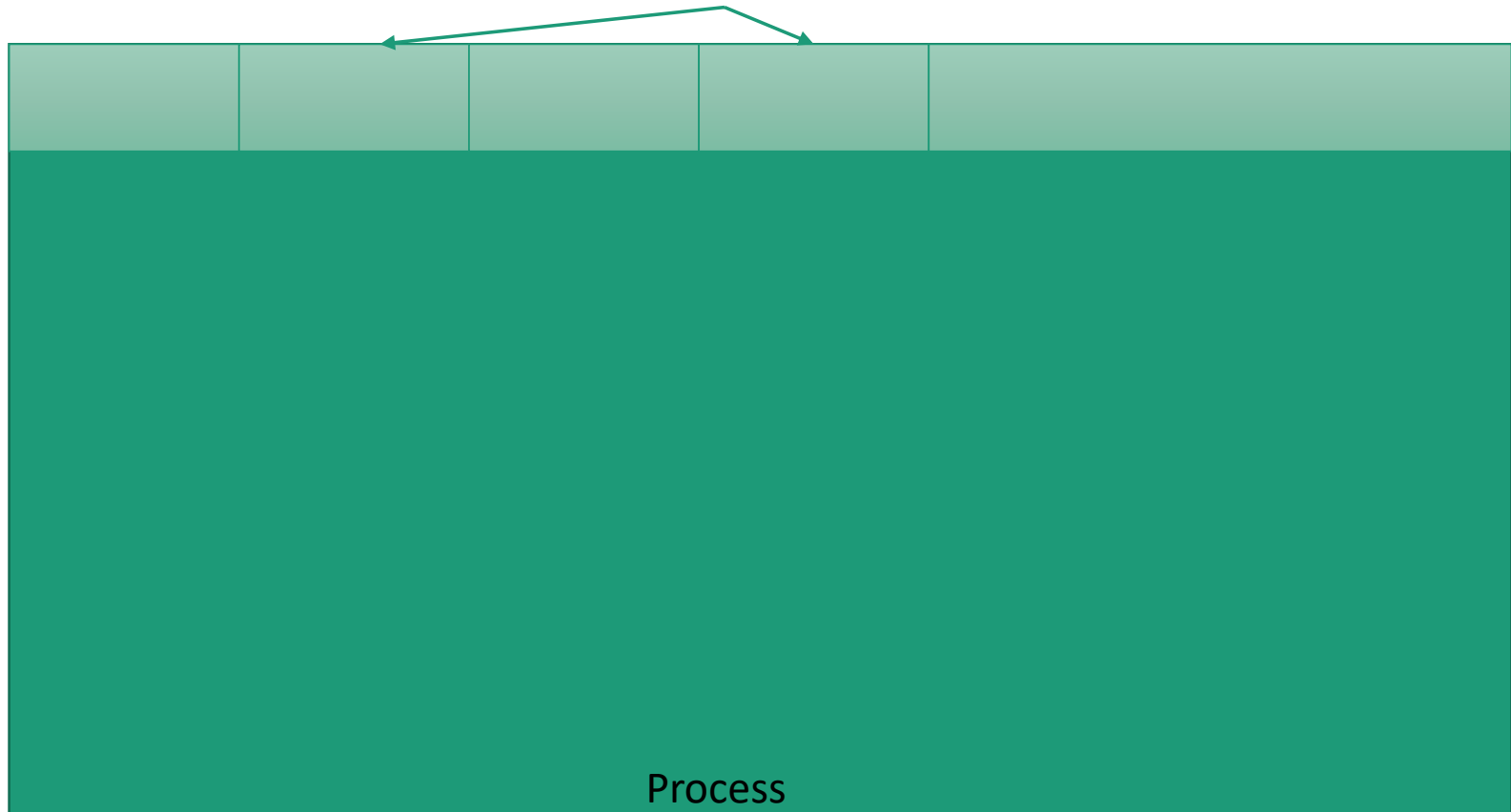
Memory Safety

Flat Memory Model



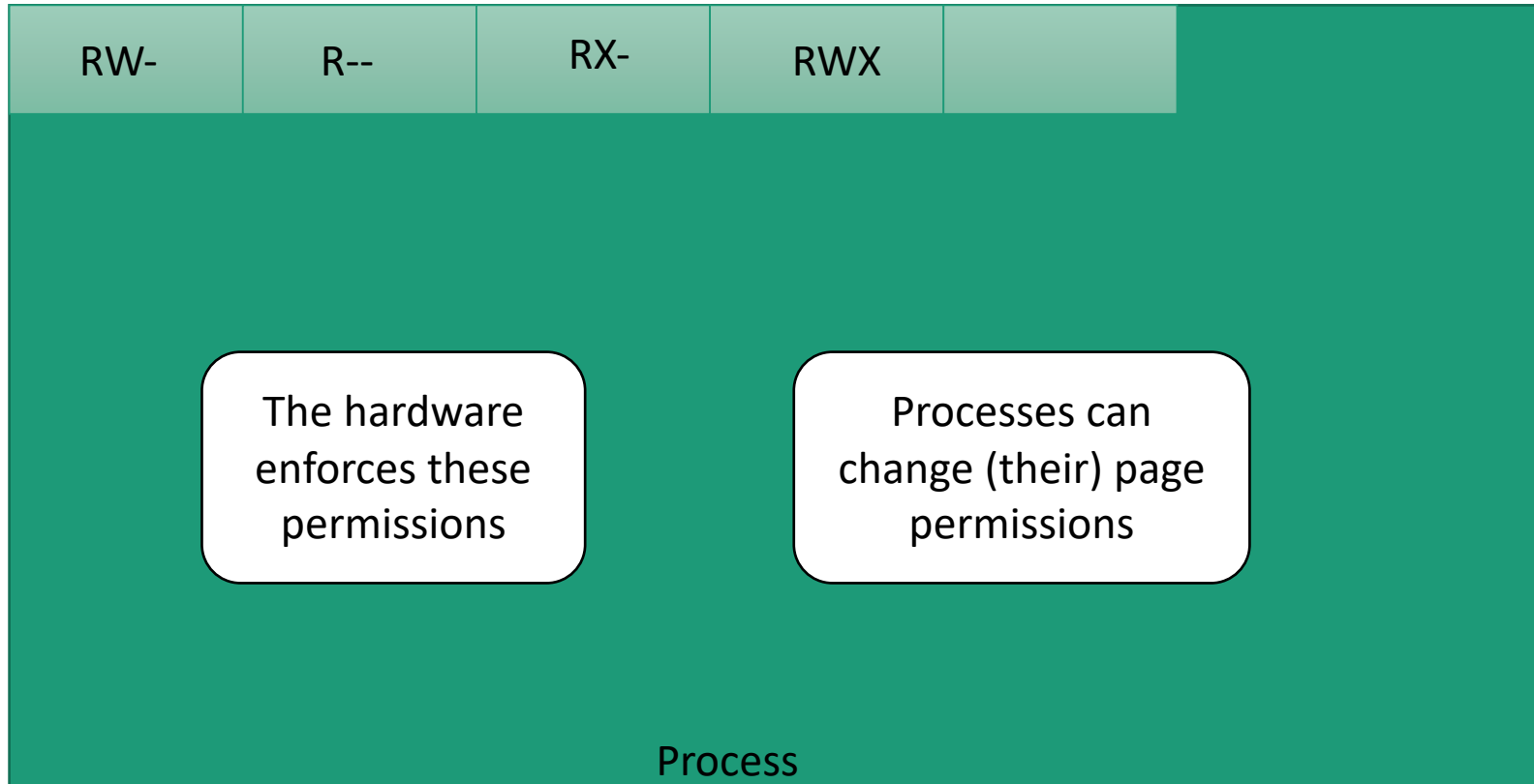
Paging and Permissions

Memory is organized in pages (usually 4KB, but can also be 2MB)



Paging and Permissions

Page permissions: readable, writable, executable



It's All Bytes

```
int n = 0xdeadbeef;  
char buffer[16] = "Hello";
```

0xef
0xbe
0xed
0xde
'H'
'e'
'l'
'l'
'o'
0x00
~~
~~



10 uninitialized bytes

It's All Bytes

```
int n = 0xdeadbeef;  
char buffer[16] = "Hello";
```

Little endian
byte order

0xef
0xbe
0xed
0xde

'H'

'e'

'l'

'l'

'o'

0x00

~~

~~

10 uninitialized
bytes

Memory Safety

Little endian
byte order

```
int n = 0xdeadbeef;  
char buffer[16] = "Hello";
```

Data are not isolated by hardware, it is up to the software

Memory safe languages introduce checks to ensure memory is not corrupted (Python, Java, etc.)

Other languages depend on the programmer to check when accessing memory (C, C+)

0xef
0xbe
0xed
0xde

'H'

'e'

'l'

'l'

'o'

0x00

~~

~~

10 uninitialized
bytes

Example

```
int simple_function(char *str)
{
    char *c;

    for (c = str; c != '\0'; c++) {
        if (*c == '0')
            *c = '1';
    }
}
```

'2'

'0'

'0'

'1'

?

?

?

~~

0x00

Reading Material

<https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>

<http://www.felixcloutier.com/x86/>

<https://en.wikibooks.org/wiki/X86> Assembly

<https://en.wikipedia.org/wiki/X86> calling conventions