How Software Executes

CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 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

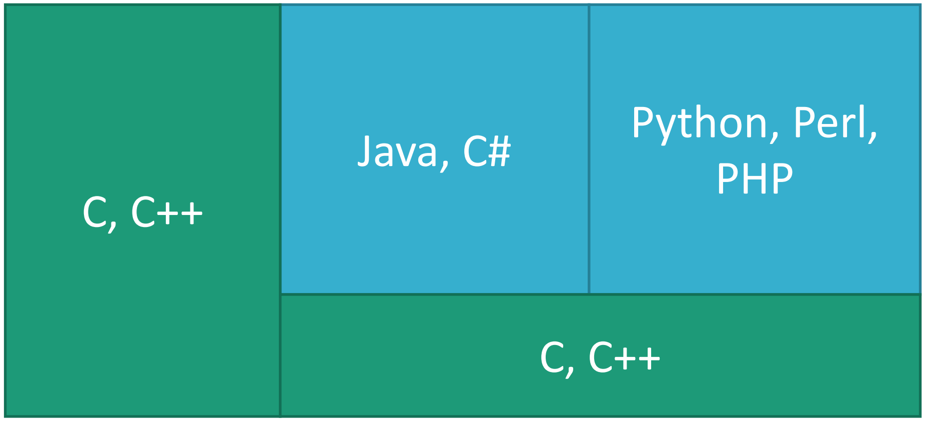
Why Do I Care?

What happens:

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

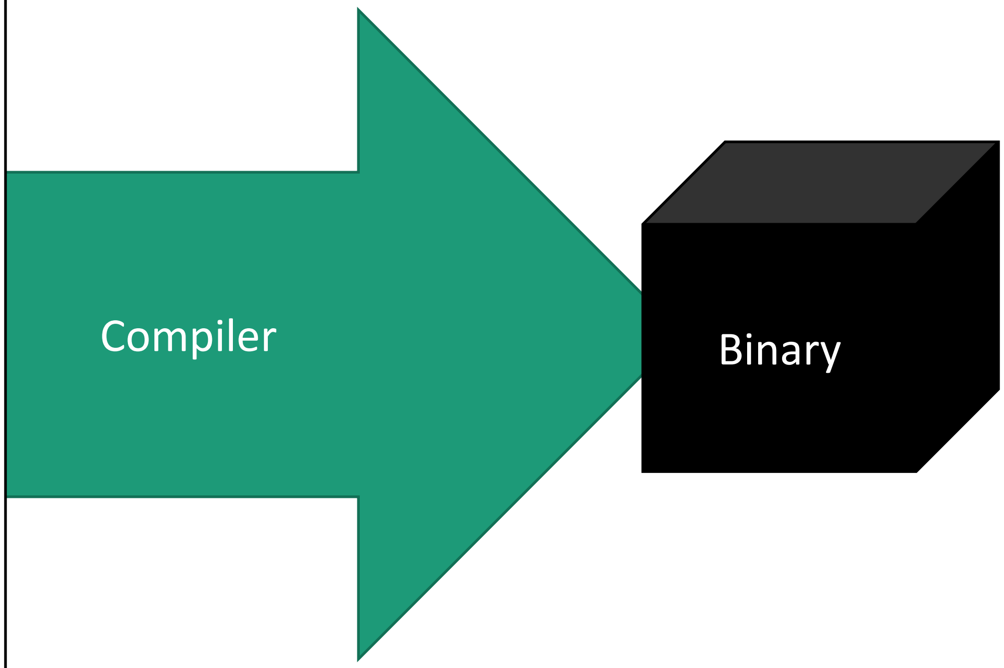
Whatever you guessed may be correct!

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 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

GAS, AT&T syntax

```
simple_function:
	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

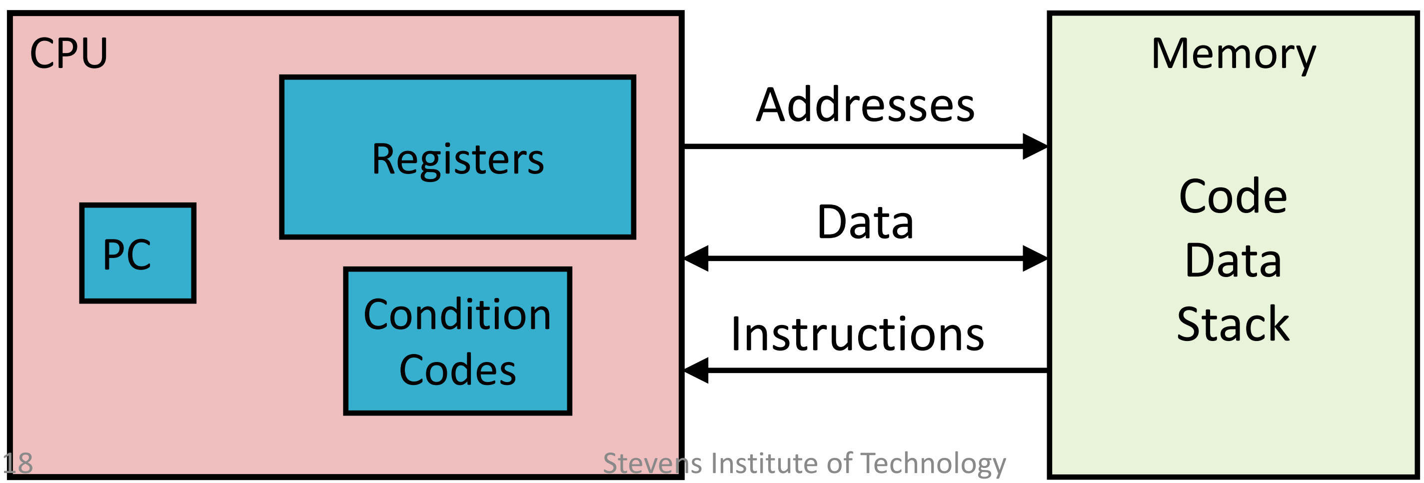
Routines

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

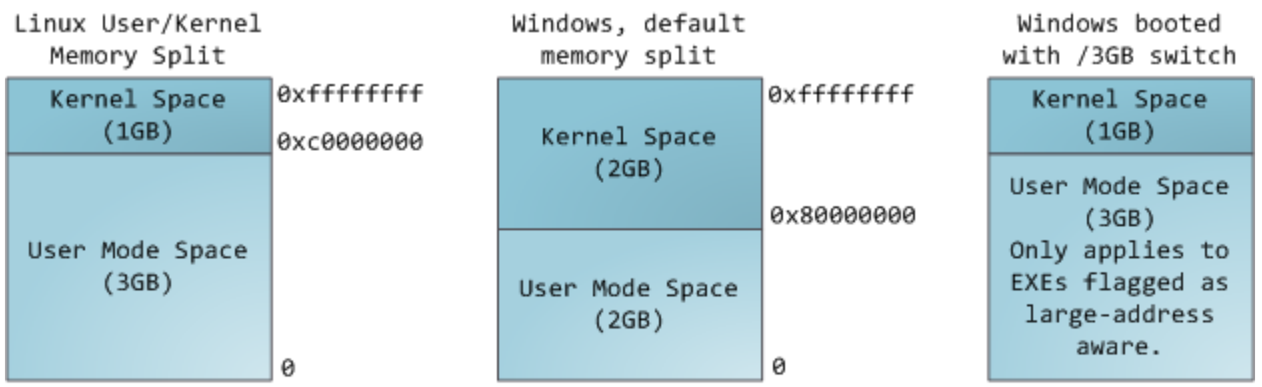
Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety

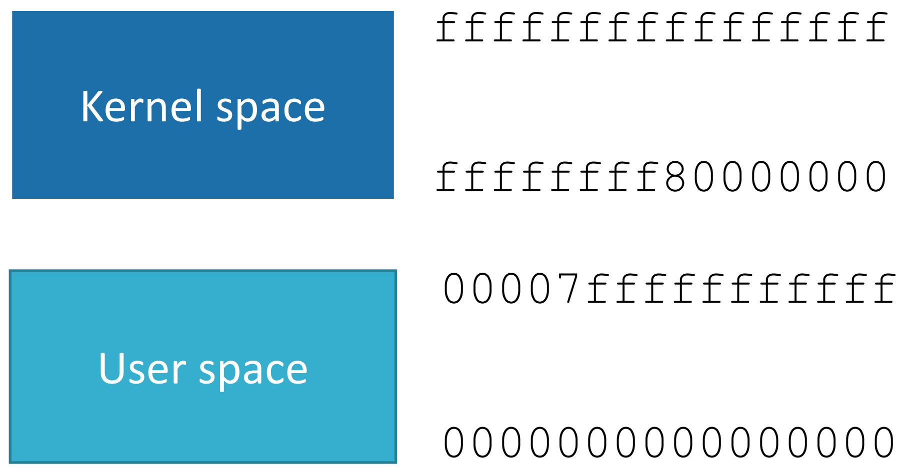
Anatomy of a Program in Memory



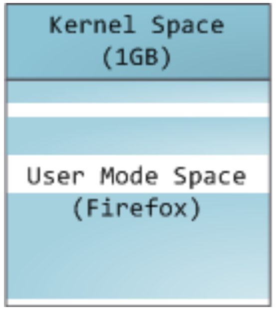
32-bit OS

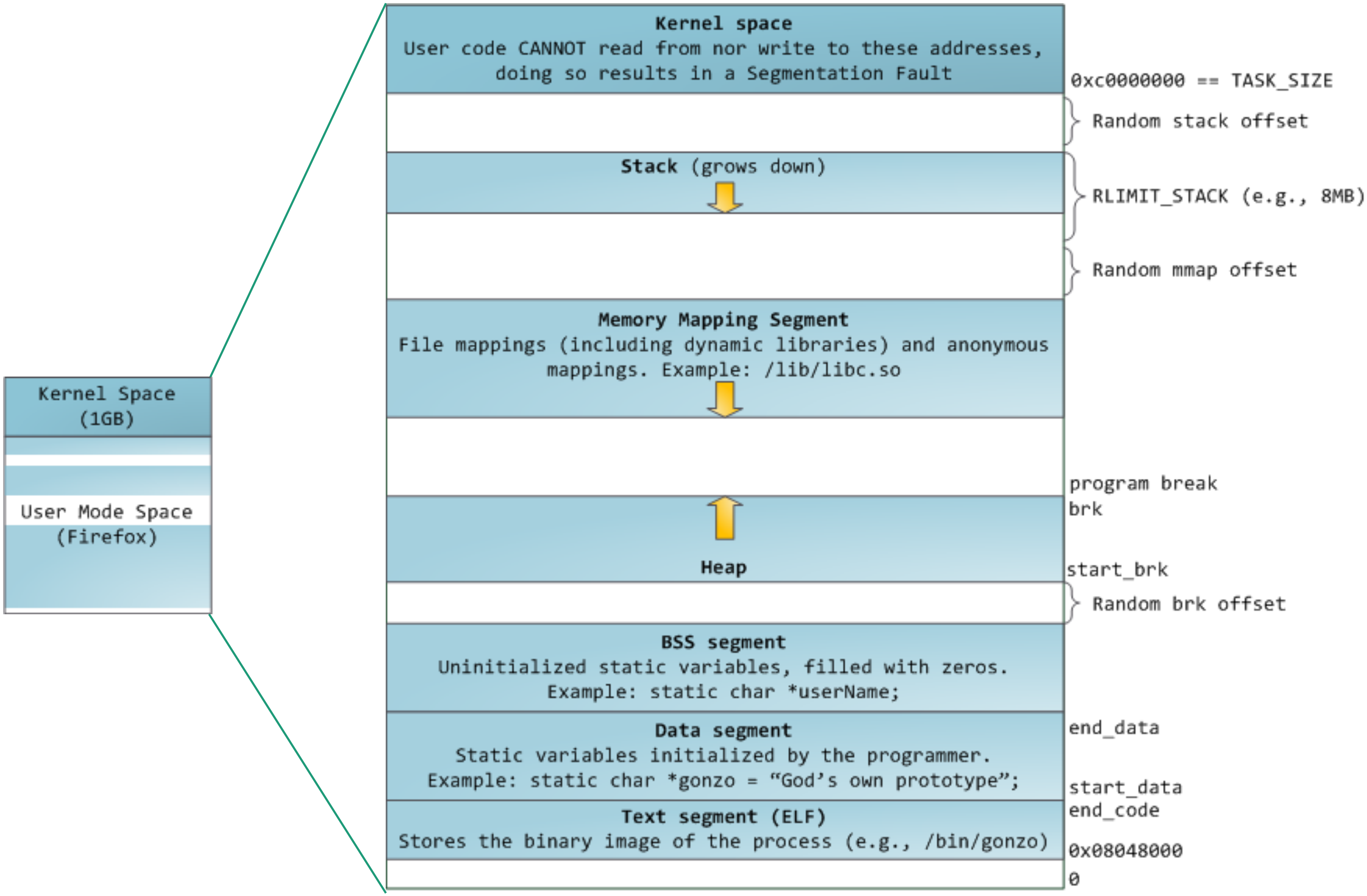
https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

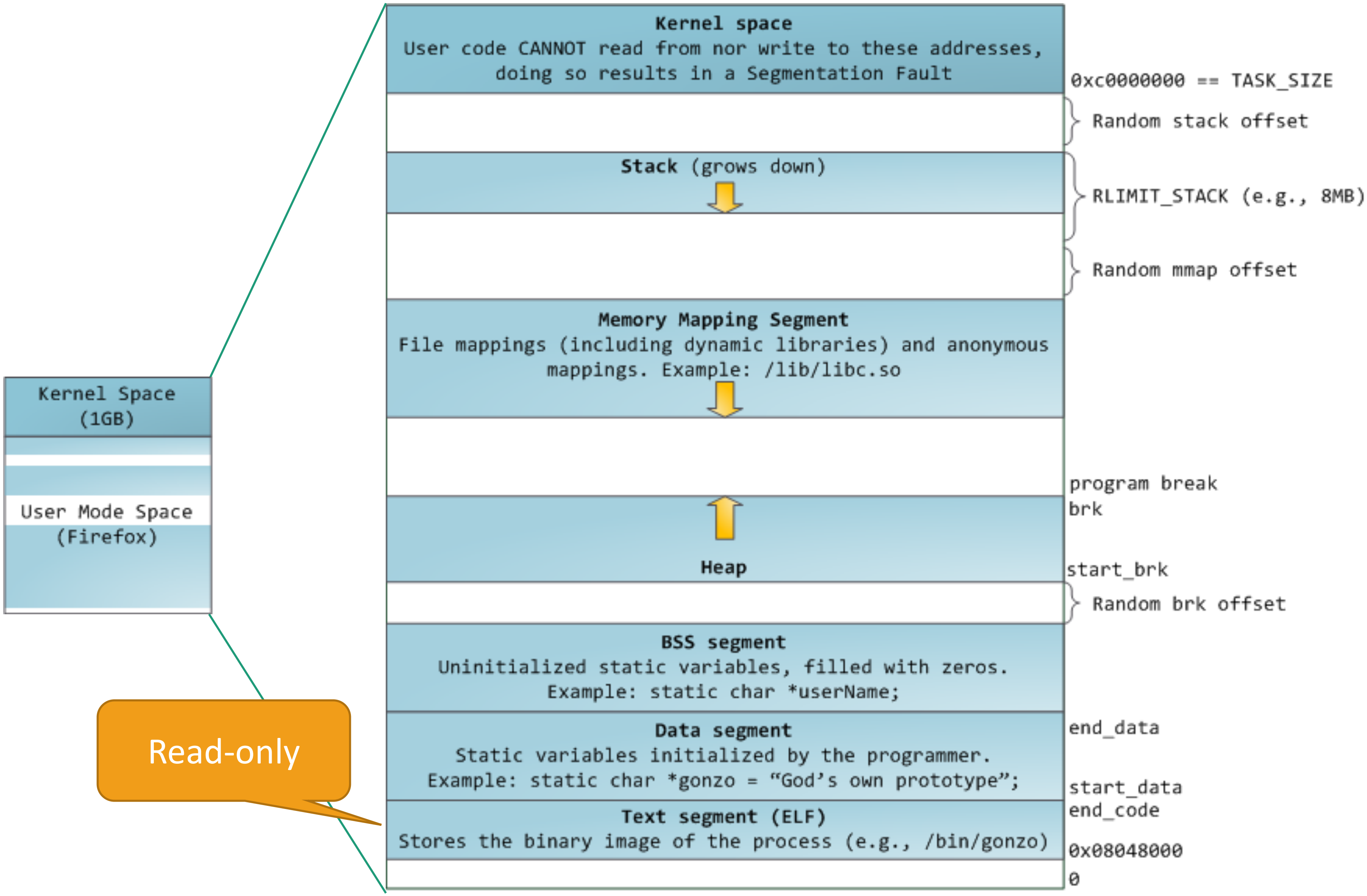
User/Kernel Space on 64-bit

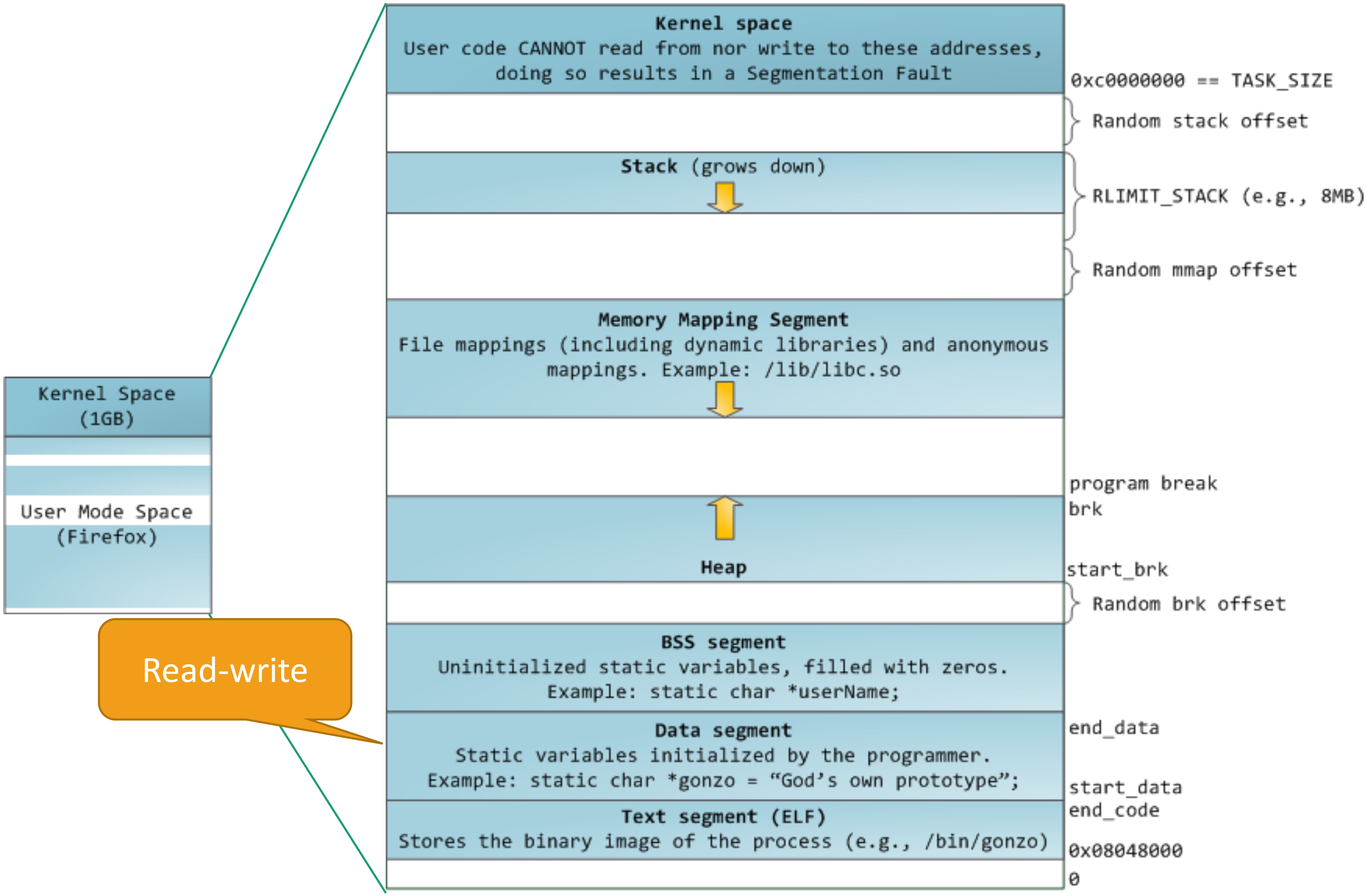


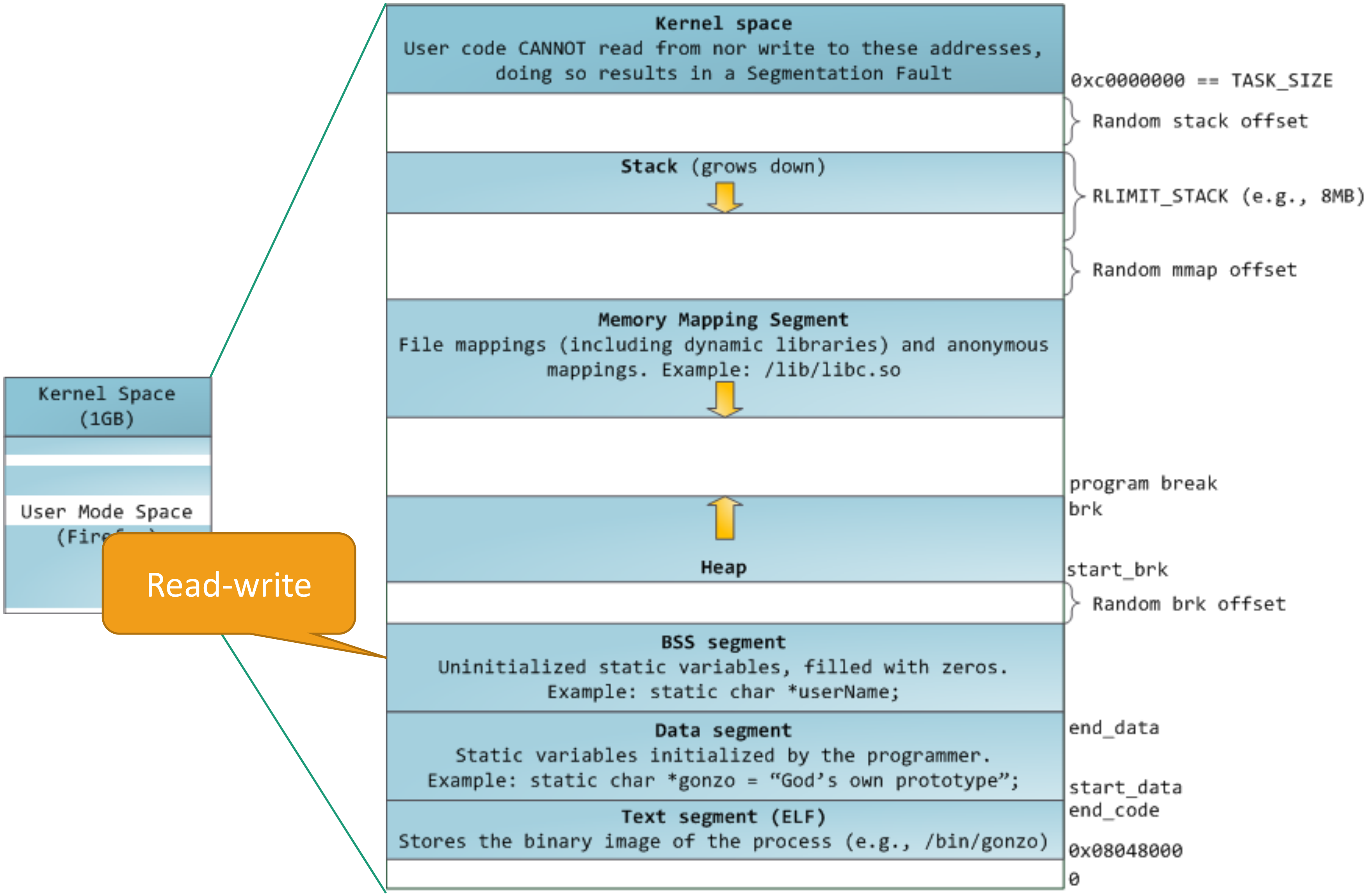
Process Anatomy



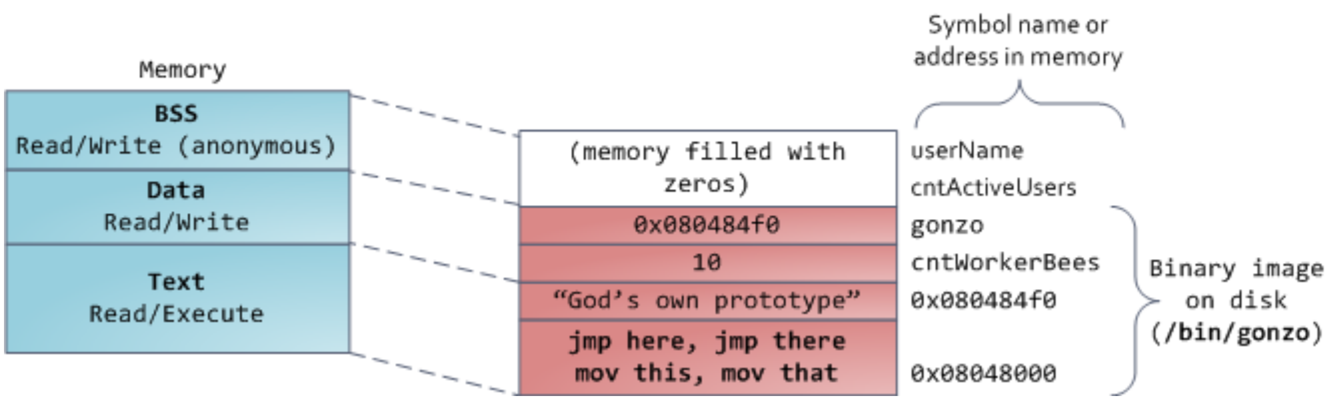

Kernel Space
(1GB)
User Mode Space
(Firefox)
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0


Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
Kernel Space (1GB)
User Mode Space (Firefox)
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0
Read-only


Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0
Kernel Space (1GB)
User Mode Space (Firefox)
Read-write


Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0
Kernel Space
(1GB)
User Mode Space
Read-write

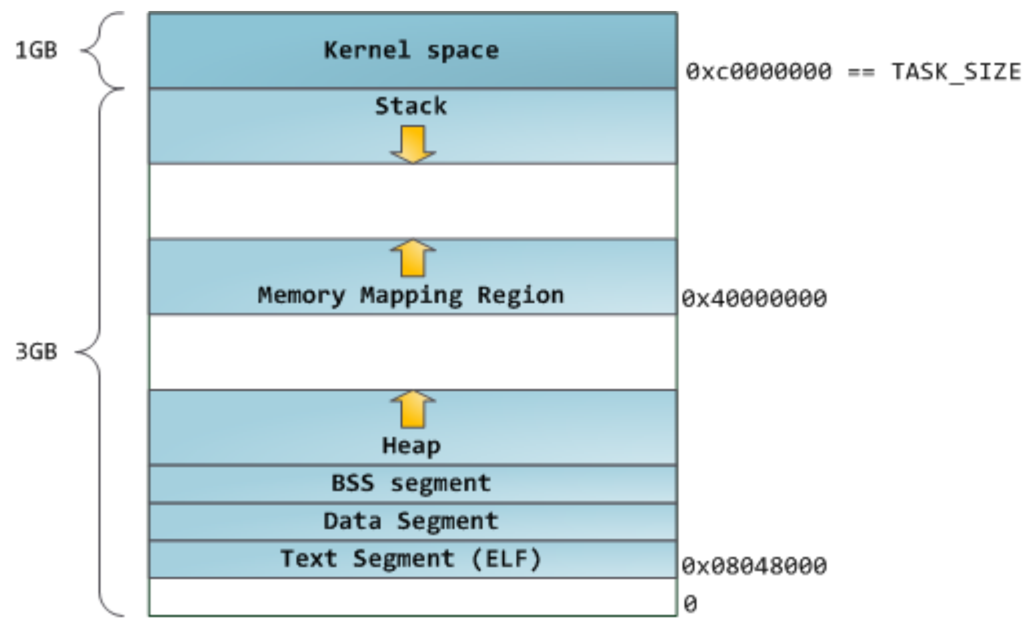
Initialized vs Uninitialized



Stack

Contains arguments, local variables, function return addresses

Read-write



Heap

Contains dynamically allocated data

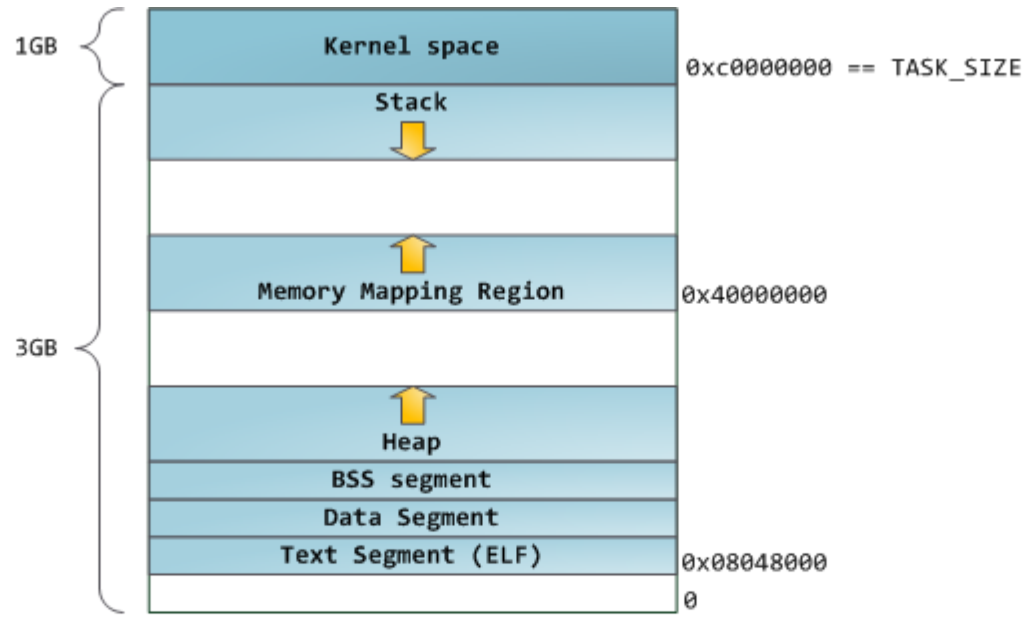
Allocated by the OS using brk() and sbrk()

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

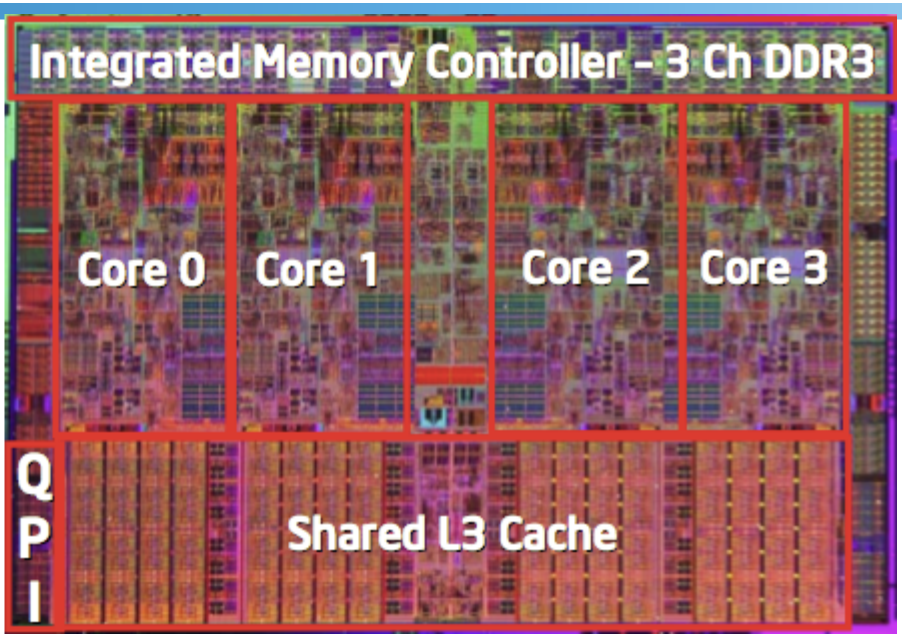
Complex instruction set computer (CISC)

- 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

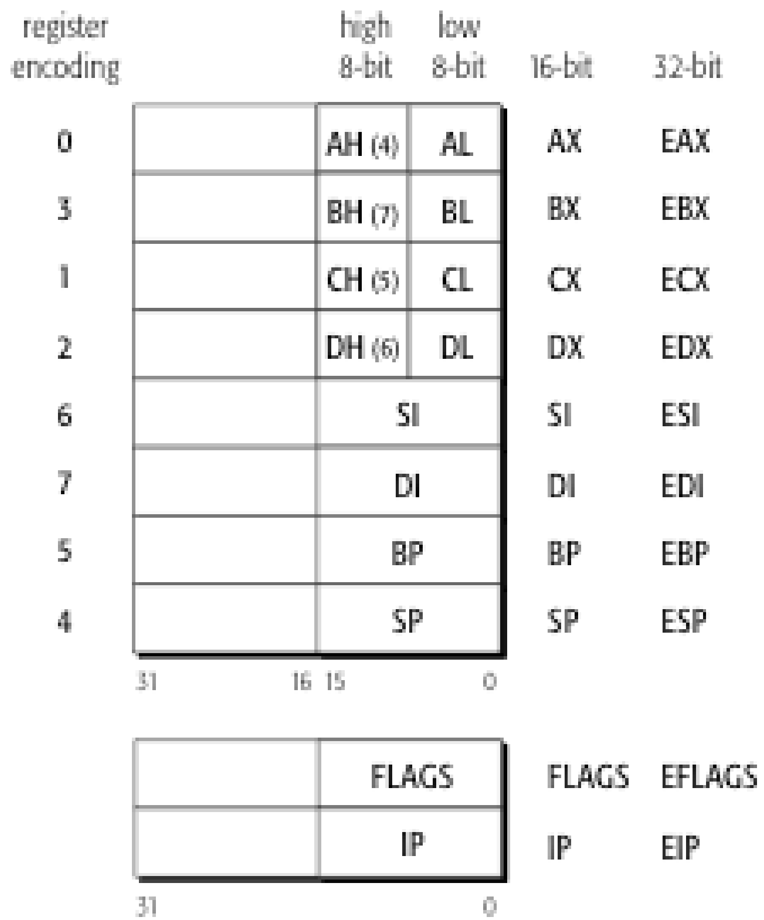
- On 32-bit architectures EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP

The instruction pointer (IP)

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

FLAGS register

- Used for control flow operations, etc.
- EFLAGS



x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%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

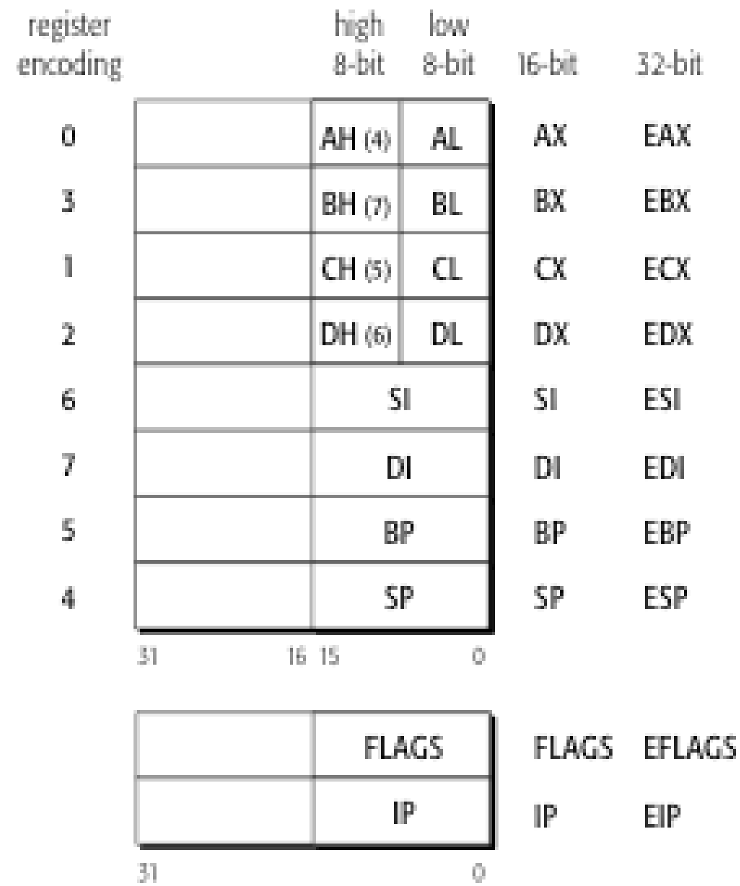
ECX: Counter for string operations and loops

EDX: I/O Operations

EDI: Destination for string operations

ESP: Stack pointer

EBP: Frame pointer



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

Instructions include a suffix that indicates the size of the operand(s)

Register operands are prefixed with a **%**

push$\boxed{\text{q}}$ %rax

Intel syntax does not include a suffix, size depends on the size of the operand

Register operands must match size
For example,

- quad → rax
- long → eax
- word → ax
- byte → ah or al

Memory Operands

Parentheses indicate a memory operand

Each memory address can be defined as:
Base+**I**ndex***S**cale+**D**isp

- 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) $\quad\quad$ $Mem[Reg[R_b]+S*Reg[R_i]+ D]$

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register
- Ri: Index register: Any, except for **`%rsp`**
- S: Scale: 1, 2, 4, or 8

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