Check-in #24

Encoding Programs

Write an X64 assembly-code program that adds the values of %rax and %rbx and exits with the sum as the exit value

Check-in #24 Solution

Encoding Programs

Announcements

Administrivia

University of Kansas | Drew Davidson

x64 Memory

Where We're At

Progress Pics

Collecting the x86 pieces needed to represent source code

Last Lecture

X64 Basics

X64 Discussion

- Some assurances

Architecture Details

- Basic instructions
- Basic memory allocation

What you should know:

- The register manipulation oprs
- The conditional jmp / cmp

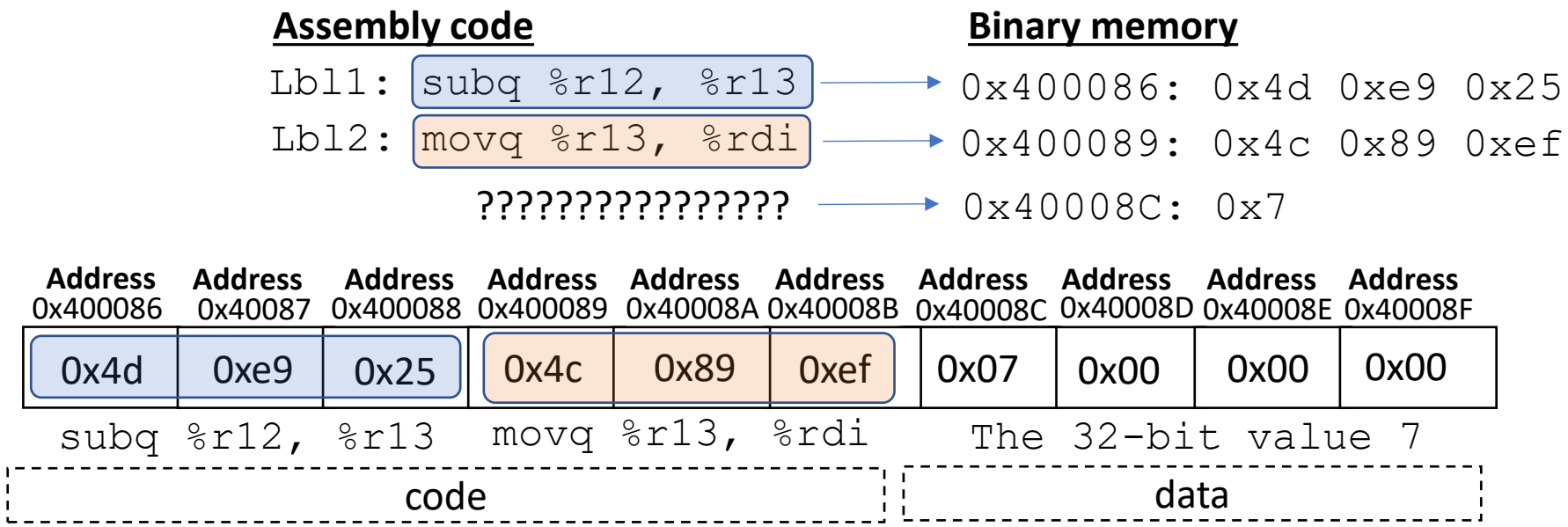
Architecture

Recall: Memory as an Array

X64 Basics

Memory is just a big ol' array of bytes (with OS mediation)

- Assembler and friends will map the code into memory
- We still need to map data to memory
  - variables, objects, strings, arrays, etc.



Memory Allocation

x64 Basics

How we allocate memory has performance implications

- Can we run out of memory at runtime?
- Is memory access fast?

Two types of allocation

- Static – memory locations pre-arranged at compile time
- Dynamic – memory locations determined at run time

Static Allocation Data Directives

x64 Basics

`.quad 13`

Statically allocate 8 bytes containing the value 13

`.byte 13`

Statically allocate 1 byte containing the value 13

`.asciz "Hello"`

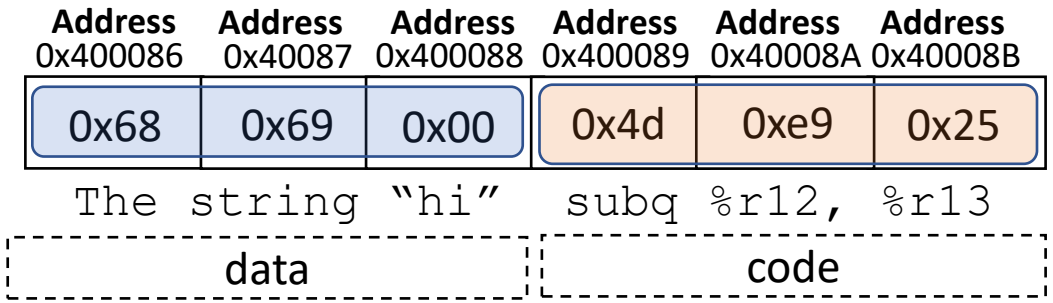
Dedicate as many bytes as needed

to fit `H e l l o \0`

Visualizing Static Allocation

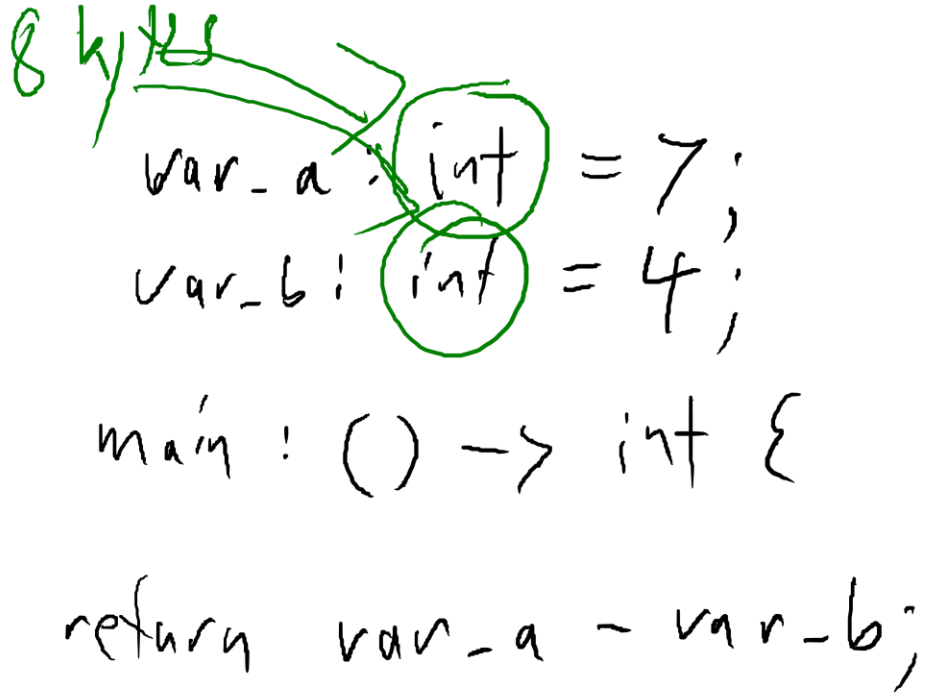
x64 Basics

```
.globl _start
.data
str: .asciiz “hi”
.text
_start:
    subq %r12, %r13
```



A Complete Program

x64 Basics



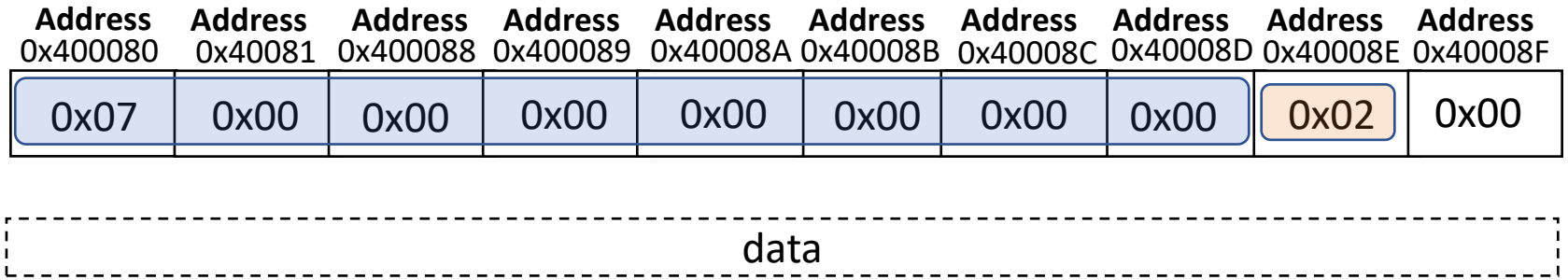

```
.globl _start
.data
global_var_a: .quad 7
global_var_b: .quad 4

.text
_start:
        #          a      b
        # r10 = r10 - r11
        movq (global_var_a), %r10
        movq (global_var_b), %r11
        subq %r11, %r10

        #exit
        movq $60, %rax
        movq %r10, %rdi
        syscall
```

Recall: Endianness

X64 Basics

x64 is *little*-endian (the little byte comes first)

```
global_var_a: .quad 7
global_var_b: .byte 2
```



Data Directives Example

X64 Basics –Memory Directives

```
.globl _start
.data
v1: .quad 7
v2: .quad 4
.text
_start:
        movq (v1), %r10
        movq (v2), %r11
        subq %r11, %r10
        #exit
        movq $60, %rax
        movq %r10, %rdi
        syscall
```

Today's Outline

Memory Layout

Memory Layout

- Static allocation
- The heap and the stack

Architecture

Visualizing Memory

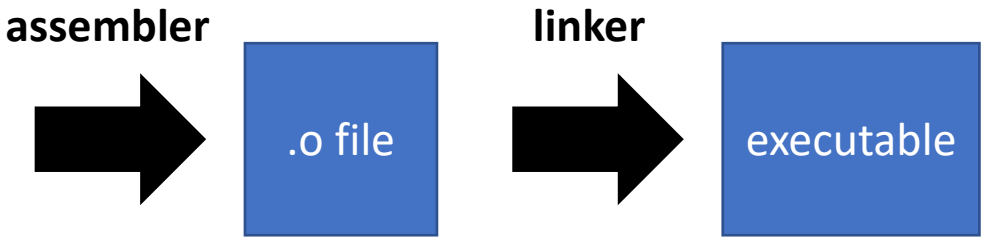
Memory Layout

```
.globl _start
.data
v1: .quad 7
v2: .quad 4
.text
_start:
        movq (v1), %r10
        movq (v2), %r11
        subq %r11, %r10
        #exit
        movq $60, %rax
        movq %r10, %rdi
        syscall
```

Visualizing Memory

Memory Layout

```
.globl _start
.data
v1: .quad 7
v2: .quad 4
.text
_start:
  movq (v1), %r10
  movq (v2), %r11
  subq %r11, %r10
  #exit
  movq $60, %rax
  movq %r10, %rdi
  syscall
```

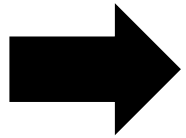


Visualizing Memory

Memory Layout

executable

objdump



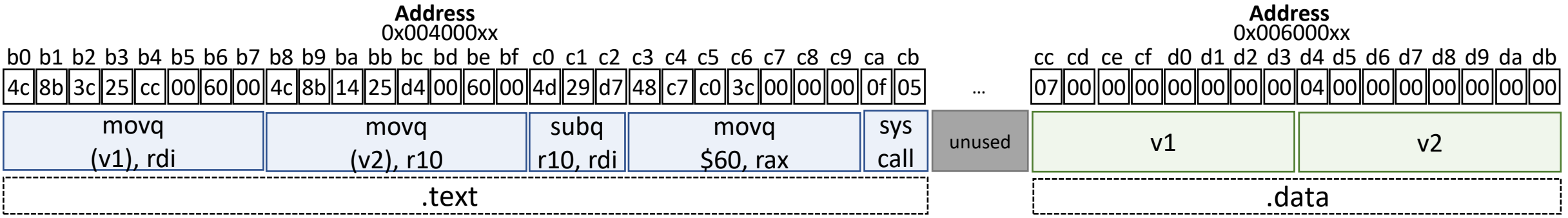
```
SYMBOL TABLE:
00000000004000b0 l    d  .text  0000000000000000 .text
00000000006000cc l    d  .data  0000000000000000 .data
00000000006000cc l       .data  0000000000000000 v1
00000000006000d4 l       .data  0000000000000000 v2
00000000004000b0 g       .text  0000000000000000 _start
00000000006000e0 g       .data  0000000000000000 _end
Disassembly of section .text:
00000000004000b0 <_start>:
  4000b0:  48 8b 3c 25 cc 00 60 00 mov    0x6000cc,%rdi
  4000b8:  4c 8b 14 25 d4 00 60 00 mov    0x6000d4,%r10
  4000c0:  4c 29 d7                sub    %r10,%rdi
  4000c3:  48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  4000ca:  0f 05                   syscall
```

Visualizing Memory

Memory Layout

```
SYMBOL TABLE:
0000000004000b0 l    d  .text  0000000000000000 .text
0000000006000cc l    d  .data  0000000000000000 .data
0000000006000cc l       .data  0000000000000000 v1
0000000006000d4 l       .data  0000000000000000 v2
0000000004000b0 g       .text  0000000000000000 _start
0000000006000e0 g       .data  0000000000000000 _end
Disassembly of section .text:
0000000004000b0 <_start>:
  4000b0:  48 8b 3c 25 cc 00 60 00 mov    0x6000cc,%rdi
  4000b8:  4c 8b 14 25 d4 00 60 00 mov    0x6000d4,%r10
  4000c0:  4c 29 d7                sub    %r10,%rdi
  4000c3:  48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  4000ca:  0f 05                   syscall
```

Let's zoom out on memory…



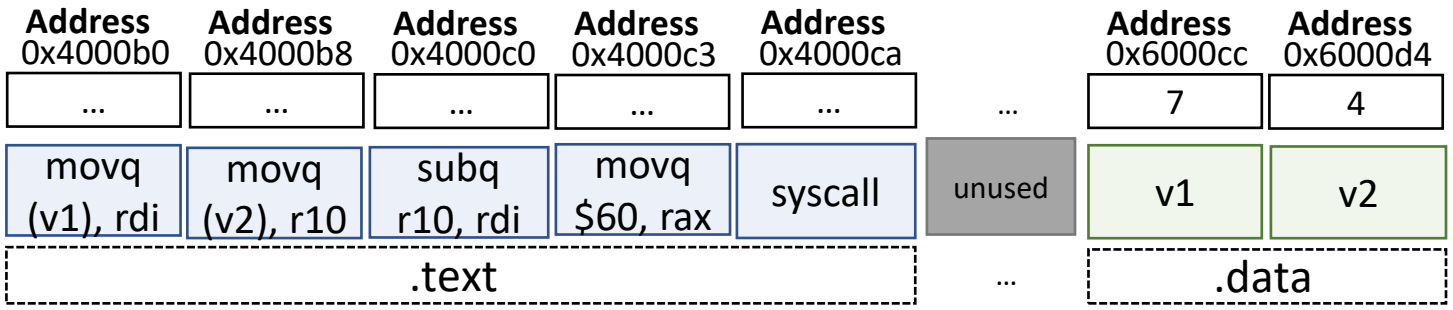
Visualizing Memory

Memory Layout

```
SYMBOL TABLE:
00000000004000b0 l    d  .text  0000000000000000 .text
00000000006000cc l    d  .data  0000000000000000 .data
00000000006000cc l       .data  0000000000000000 v1
00000000006000d4 l       .data  0000000000000000 v2
00000000004000b0 g       .text  0000000000000000 _start
00000000006000e0 g       .data  0000000000000000 _end
```

```
Disassembly of section .text:
00000000004000b0 <_start>:
  4000b0:  48 8b 3c 25 cc 00 60 00 mov    0x6000cc,%rdi
  4000b8:  4c 8b 14 25 d4 00 60 00 mov    0x6000d4,%r10
  4000c0:  4c 29 d7                sub    %r10,%rdi
  4000c3:  48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  4000ca:  0f 05                   syscall
```

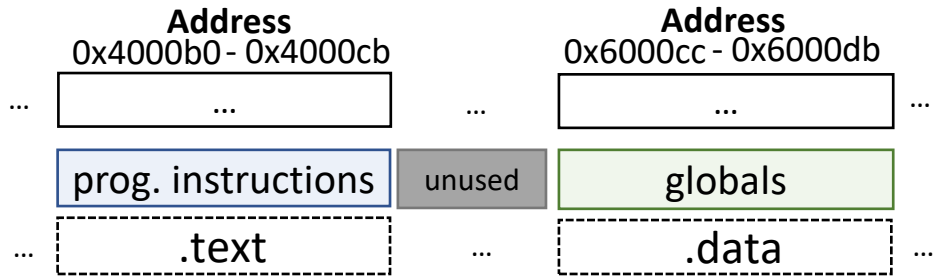
Let's zoom out on memory some more…



Visualizing Memory

Memory Layout

```
SYMBOL TABLE:
0000000004000b0 l    d  .text  0000000000000000 .text
0000000006000cc l    d  .data  0000000000000000 .data
0000000006000cc l       .data  0000000000000000 v1
0000000006000d4 l       .data  0000000000000000 v2
0000000004000b0 g       .text  0000000000000000 _start
0000000006000e0 g       .data  0000000000000000 _end
Disassembly of section .text:
0000000004000b0 <_start>:
  4000b0:  48 8b 3c 25 cc 00 60 00 mov    0x6000cc,%rdi
  4000b8:  4c 8b 14 25 d4 00 60 00 mov    0x6000d4,%r10
  4000c0:  4c 29 d7                sub    %r10,%rdi
  4000c3:  48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  4000ca:  0f 05                   syscall
```



Static Allocation

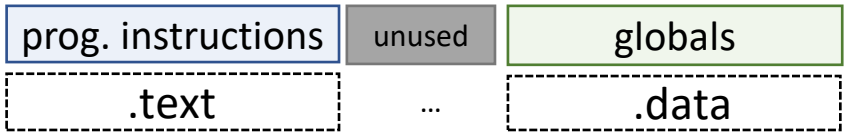
Memory Layout

This layout scheme is known as static allocation (*why?*):

- One fixed-size segment for all data
- One fixed-sized segment for all program instructions

Static allocation is sufficient for some simple languages

- e.g, FORTRAN I (1957)

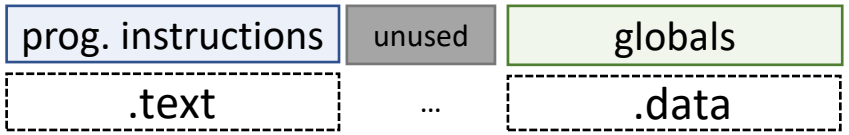


Static Allocation

Memory Layout

As languages evolve, static allocation still sufficient…

- Subroutines
  - Like functions with no callees



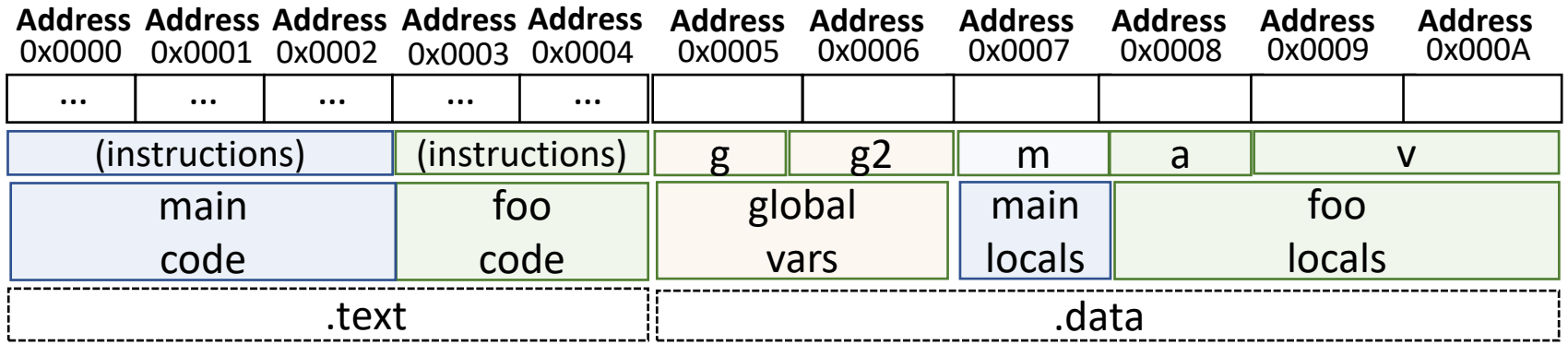
Static Allocation

Memory Layout

As languages evolve, static allocation still sufficient…

- Subroutines
  - Like functions with no callees

```
byte g;
byte g2;
foo(int16 v){
    byte a;
    …
}
main(){
    byte m;
    …
}
```

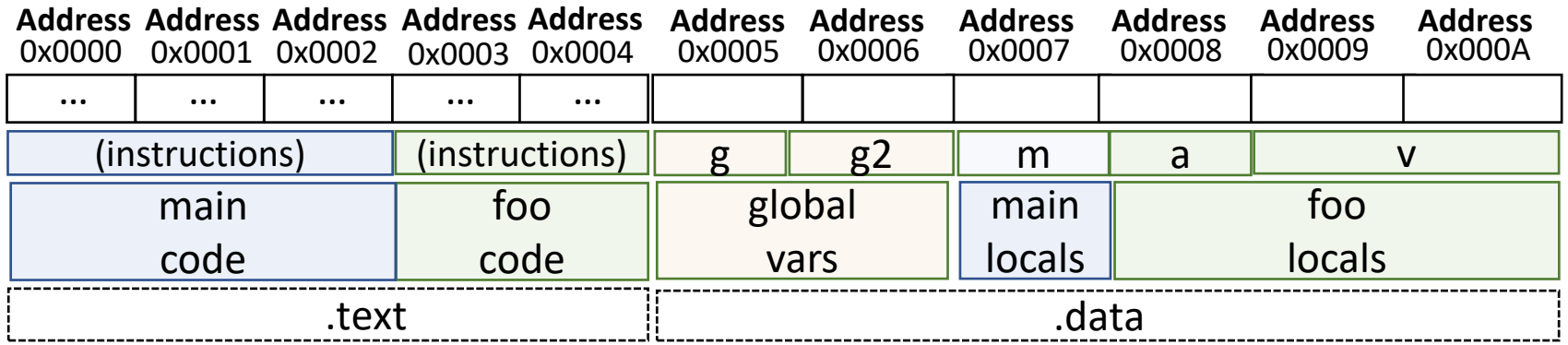


Static Allocation

Memory Layout

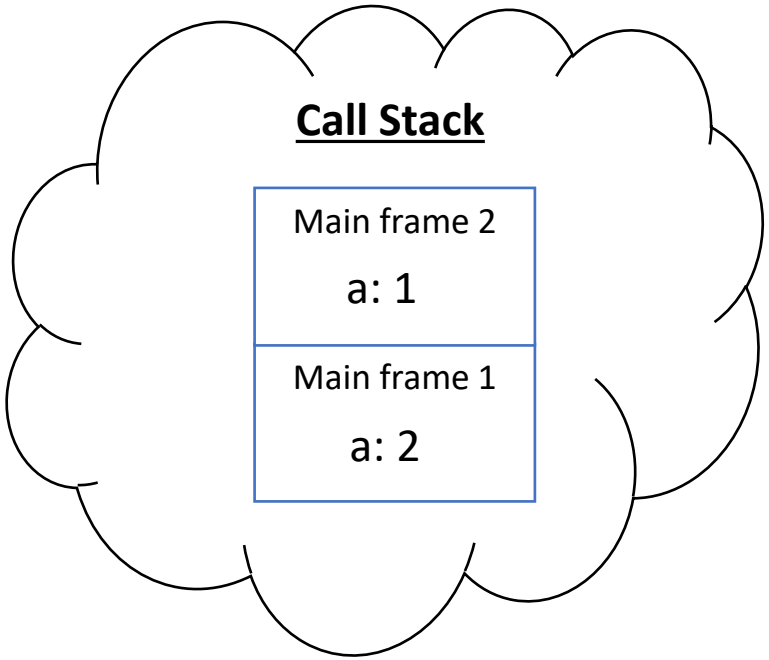
Some languages features are difficult/impossible to support with purely-static allocation

- Recursion

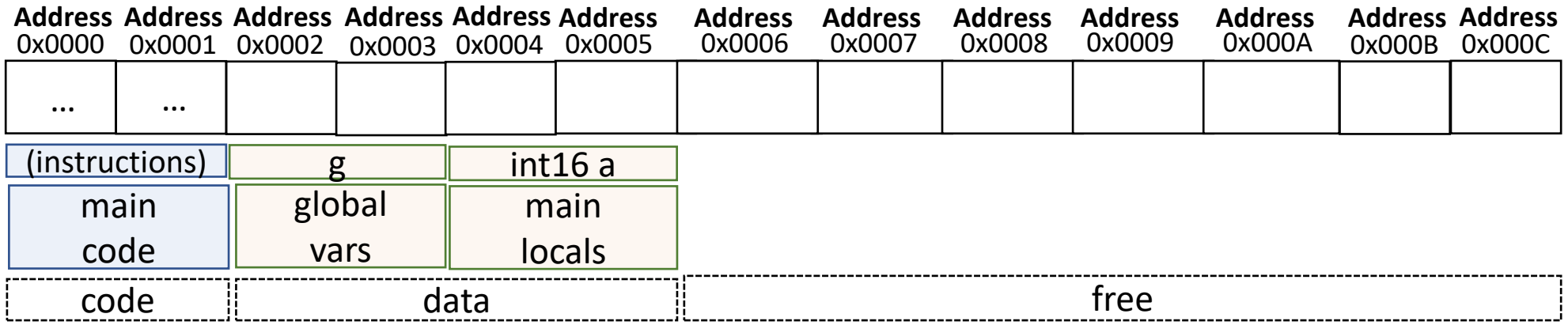


The Recursion Problem

Beyond Static Allocation




```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    cout << a;
}
```



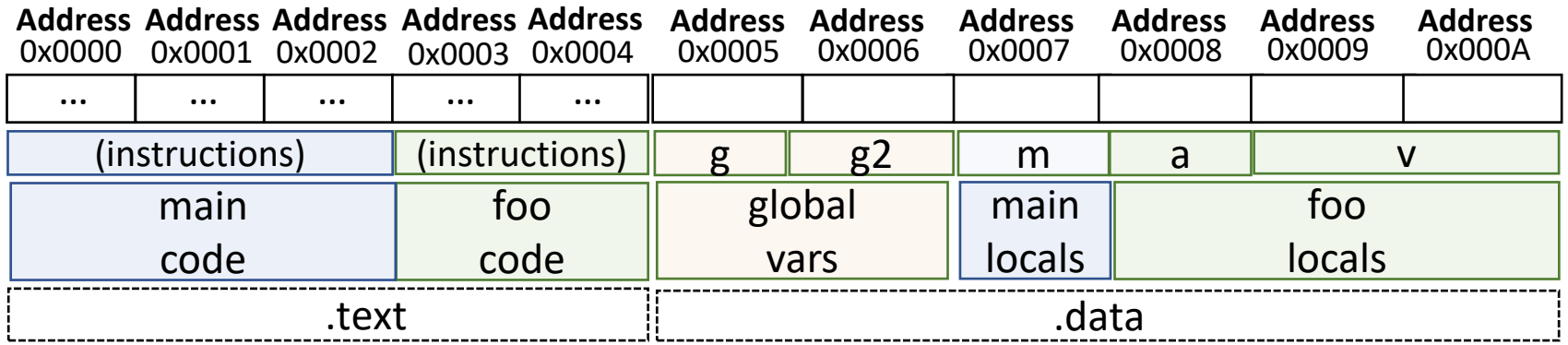
Static Allocation

Memory Layout

Some languages features are difficult/impossible to support with purely-static allocation

- Recursion
- Runtime-allocated objects
  - Can’t know, apriori, how much memory we’ll need
  - Would be super wasteful to just allocate the entire space

We need new memory segments for these features



Today's Outline

Memory Layout

Memory Layout

- Static allocation
- The heap and the stack

Architecture

The Heap and the Stack: Capabilities

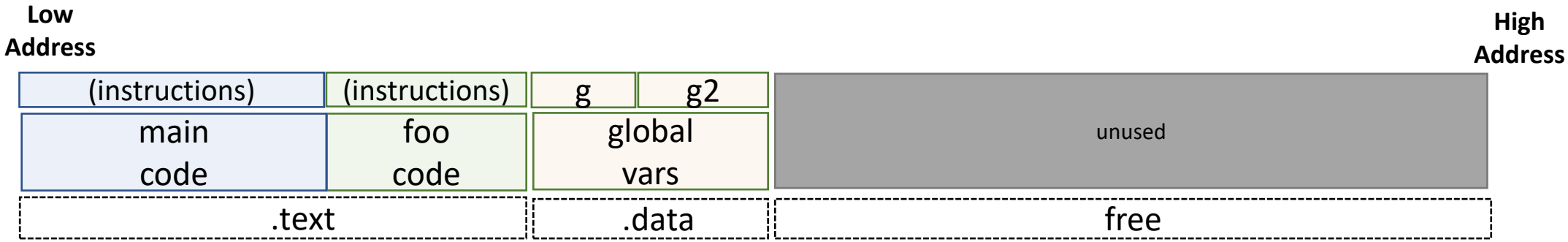
Memory Layout

The heap

- Stores malloc'ed data
- Segment size managed at runtime (via syscall to OS)

The stack

- Holds function invocations (1 per call)
- Fixed size, but starts off with lots of free space



The Heap and the Stack: Capabilities

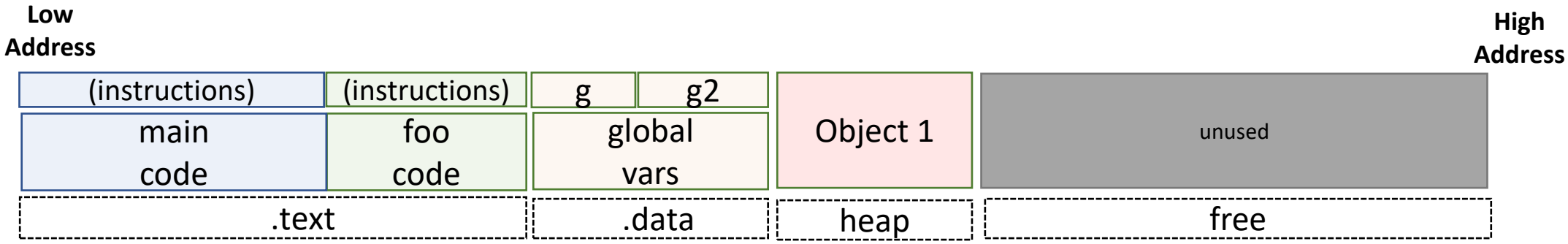
Memory Layout

The heap

- Stores malloc'ed data
- Segment size managed at runtime (via syscall to OS)

The stack

- Holds function invocations (1 per call)
- Fixed size, but starts off with lots of free space



The Heap and the Stack: Capabilities

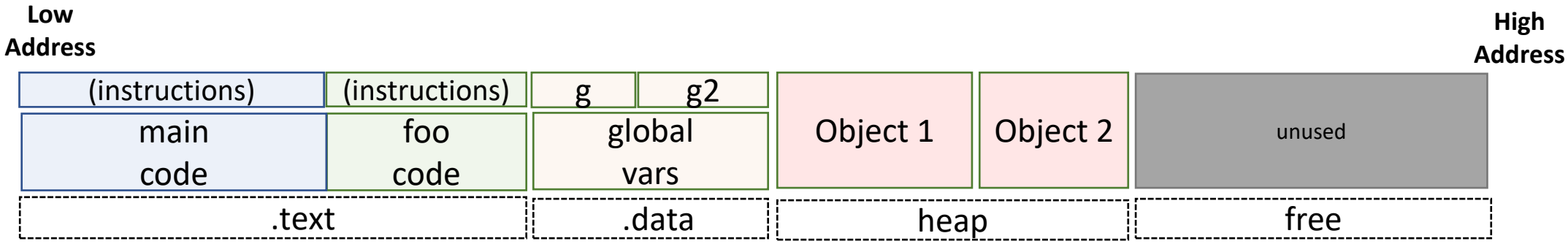
Memory Layout

The heap

- Stores malloc'ed data
- Segment size managed at runtime (via syscall to OS)

The stack

- Holds function invocations (1 per call)
- Fixed size, but starts off with lots of free space



The Heap and the Stack: Capabilities

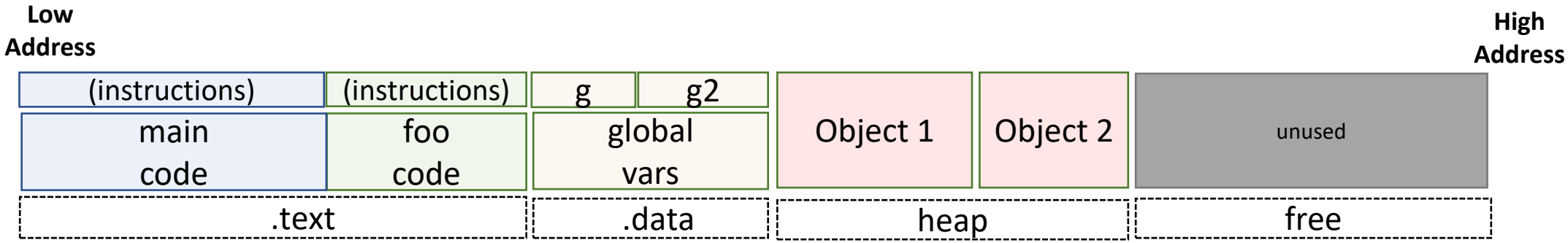
Memory Layout

The heap

- Stores malloc'ed data
- Segment size managed at runtime (via syscall to OS)

The stack

- Holds function invocations (1 per call)
- Fixed size, but starts off with lots of free space



The Heap and the Stack: Capabilities

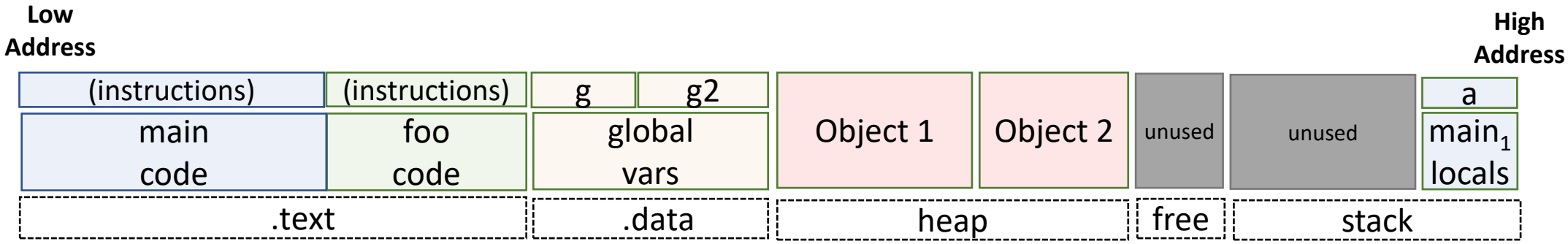
Memory Layout

The heap

- Stores malloc'ed data
- Segment size managed at runtime (via syscall to OS)

The stack

- Holds function invocations (1 per call)
- Fixed size, but starts off with lots of free space



The Heap and the Stack: Capabilities

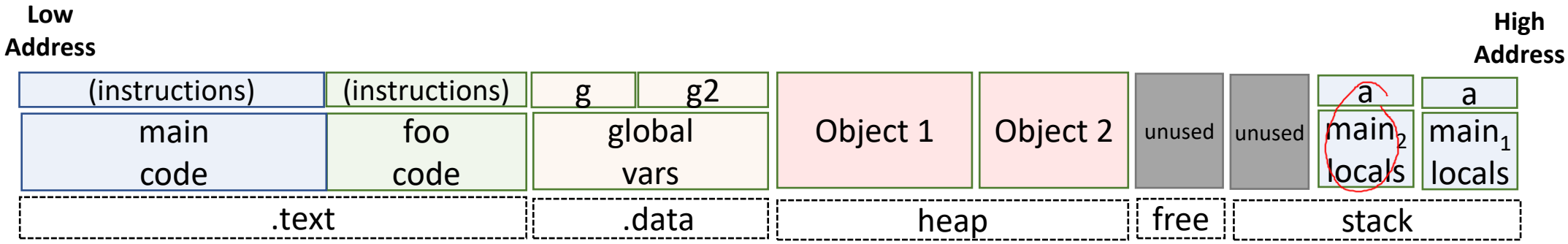
Memory Layout

The heap

- Stores malloc'ed data
- Segment size managed at runtime (via syscall to OS)

The stack

- Holds function invocations (1 per call)
- Fixed size, but starts off with lots of free space



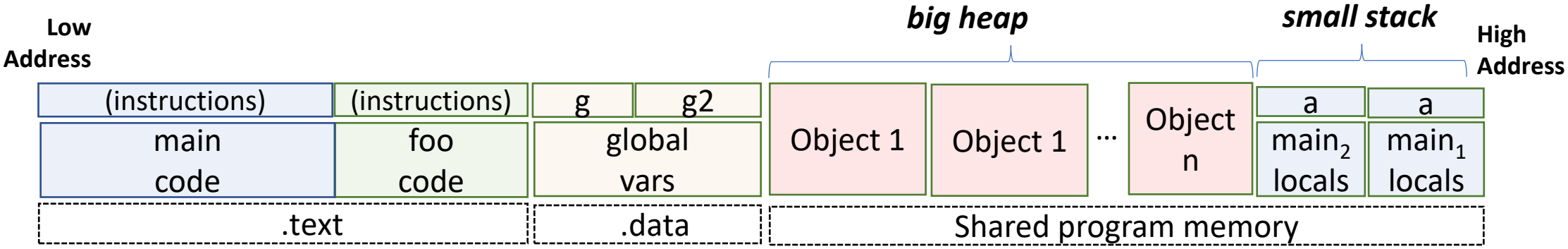
Note: the Stack grows DOWN

Memory Layout

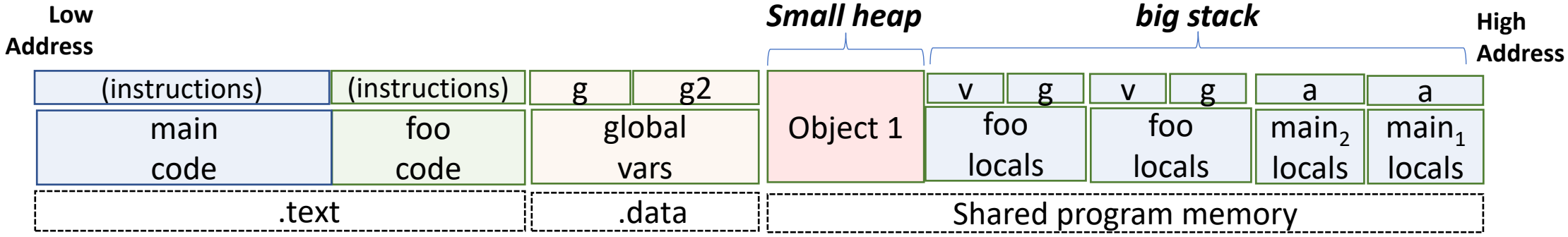
For historical reasons

- Needed to share limited memory space

vintage memory diagram, scenario #1



vintage memory diagram, scenario #2



Summary: "Modern" Memory Allocation

Memory Layout

Instruction code

- Static allocation in the .text section

Global variables

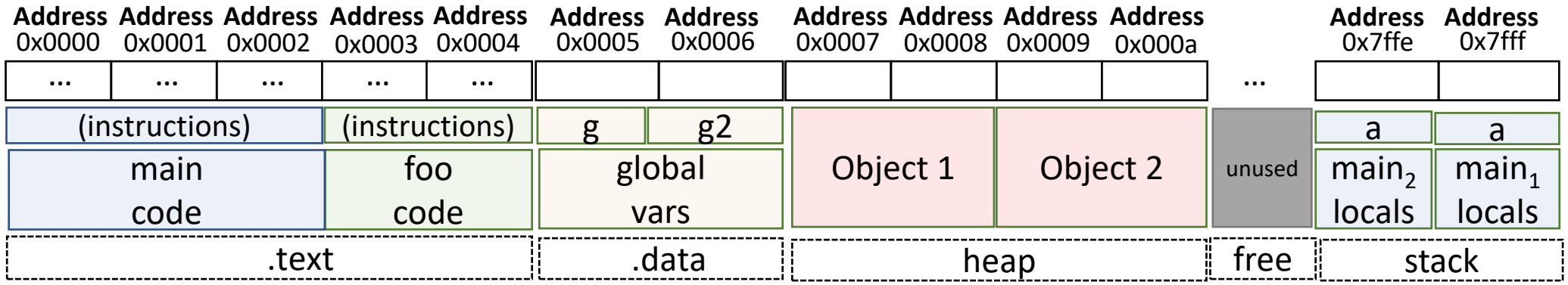
- Static allocation in the .data section

Malloc'ed data

- Dynamic allocation on the heap

Local variables

- Dynamic allocation on the stack



Lecture Done!

Next Time: x64 Practice

Next Time

- Handling more complicated programs in x64