Discussion 02

C/Memory

Aditya Balasubramanian

`aditbala [at] berkeley [dot] edu`

Announcements 📣

Agenda

- C Basics
- Structs and Unions
- Memory Review
- Endianness
- Q&A

C Basics 💻

Types in C

- Everything in the computer is just bits!
    - What does that even mean?
- Definitions
    - char := 1 byte
    - short := 2 bytes
    - int := 4 bytes
    - long := 8 bytes
- Pointers? “32/64-bit systems”

Pointers (Pt. 1)

- Variables
  - nicknames for a value
  - Must be declared, and then initialized
    - `char *x; // declaration`
    - `x = "hello" // initialization`
  - Sit somewhere in memory
  - Are a series of `n` bits interpreted in a specific way
- Memory Addresses
  - Are just another variable type, that looks a bit different (like arrays)
  - Technically, just integers!
  - Declared with `type*`, then initialized with a valid memory address value

Pointers (Pt. 2)

- Dereference
  - Memory addresses hold a value, that can be retrieved by `*addr`
- Address
  - Every variable has its own memory address, found at `&var`! (where the variable lives)
- Pointers are sized depending on the architecture!
- Why do pointer sizes change?

Useful Tips

```c
char **double_ptr = malloc(sizeof(char*) * num_elem);
char *single_ptr = malloc(sizeof(char) * num_elem);
double_ptr[1] = single_ptr;
*double_ptr = single_ptr;
double_ptr = &single_ptr;
```

The `sizeof()` function

- Returns the size of the type of the given variable, in bytes
- What’s sizeof(char*)?
  - 1 byte

Pointer Arithmetic

```
int *int_arr = malloc(20 * sizeof(int))
```

- `int_arr[i]` vs. `*(int_arr + i)`
- What happens when you add `i` to a pointer?

Structs and Unions

Structs

- Blocks of memory storing consecutive values
- Accessing fields
  - `struct.field`
  - `(*struct_ptr).field`
  - `struct_ptr->field`
- Compiler aligns memory to field’s natural size
  - Every field sits on memory addresses that is a multiple of its size

TypeDef

- Create a nickname that can be referred to for values
- Typedef vs. `#define`
  - Both give aliases/nicknames to values
  - Typedef can only give symbolic names
  - #define can define aliases for values
    - `#define ONE 1;`
    - `typedef unsigned char BYTE;`
  - Typedef: interpreted by compiler
  - `#define`: substituted by the C pre-processor

TypeDef (Example)

```
struct ll_node {
        int val = 0;
        ll_node* next;
};
struct ll_node {
        int val = 0;
        Node* next;
} Node;
```

Unions

- Effectively the same as structs EXCEPT not all fields will exist simultaneously
- Can still use dot notation to get fields
- Memory will be allocated for the largest field in the union!

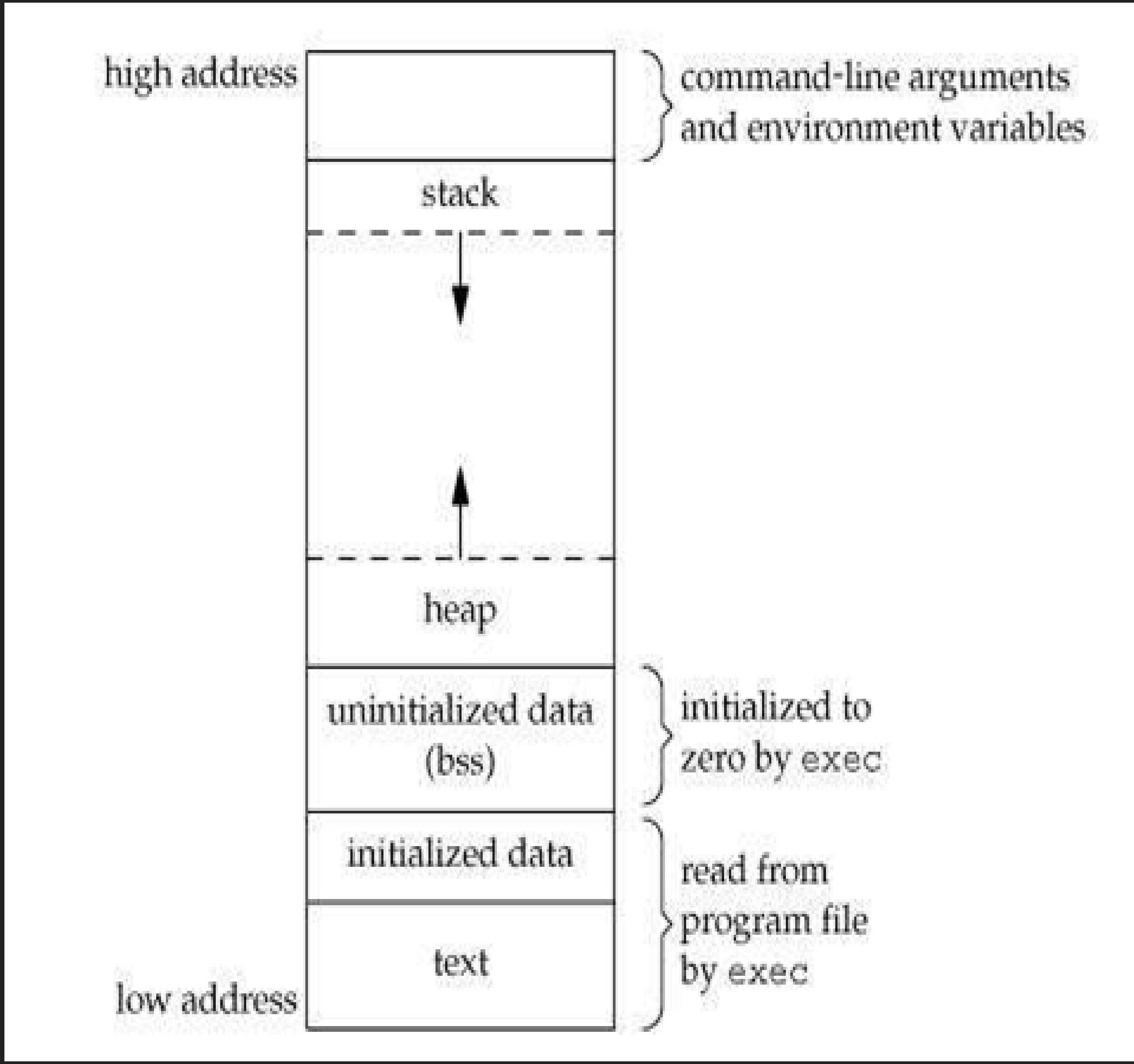
Memory Structure

Memory Structure

- Memory is contiguous!
- From top-down, it’s separated into 4 chunks
  - Stack
  - Heap
  - Static/data
  - Text/Code

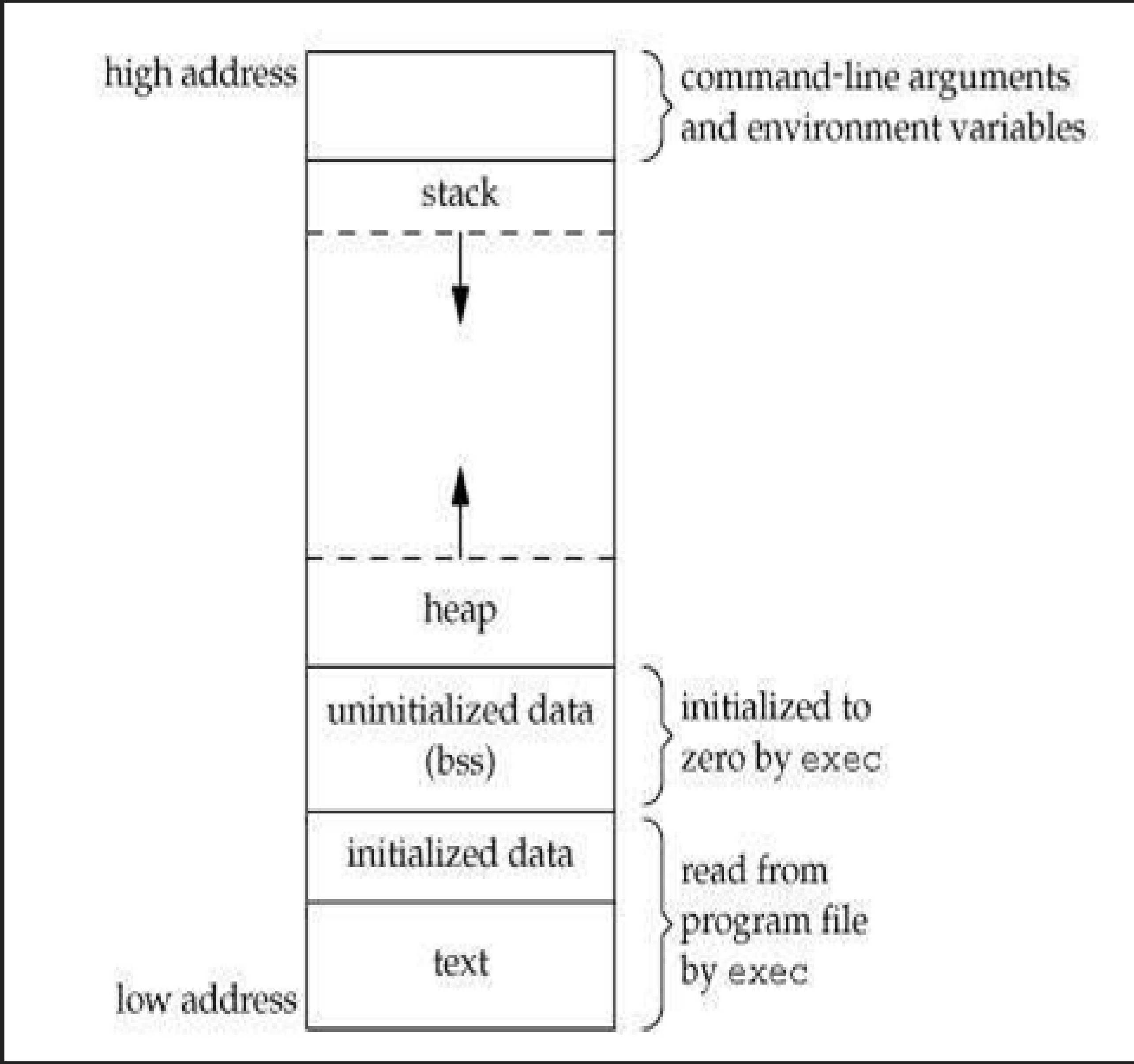
Memory Structure (Pt.1)

- Code/text
  - The code that you intend to execute
- Static/data
  - Primarily constants that don't need to be changed CAN be changed, e.g. global variables


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Memory Structure (Pt.2)

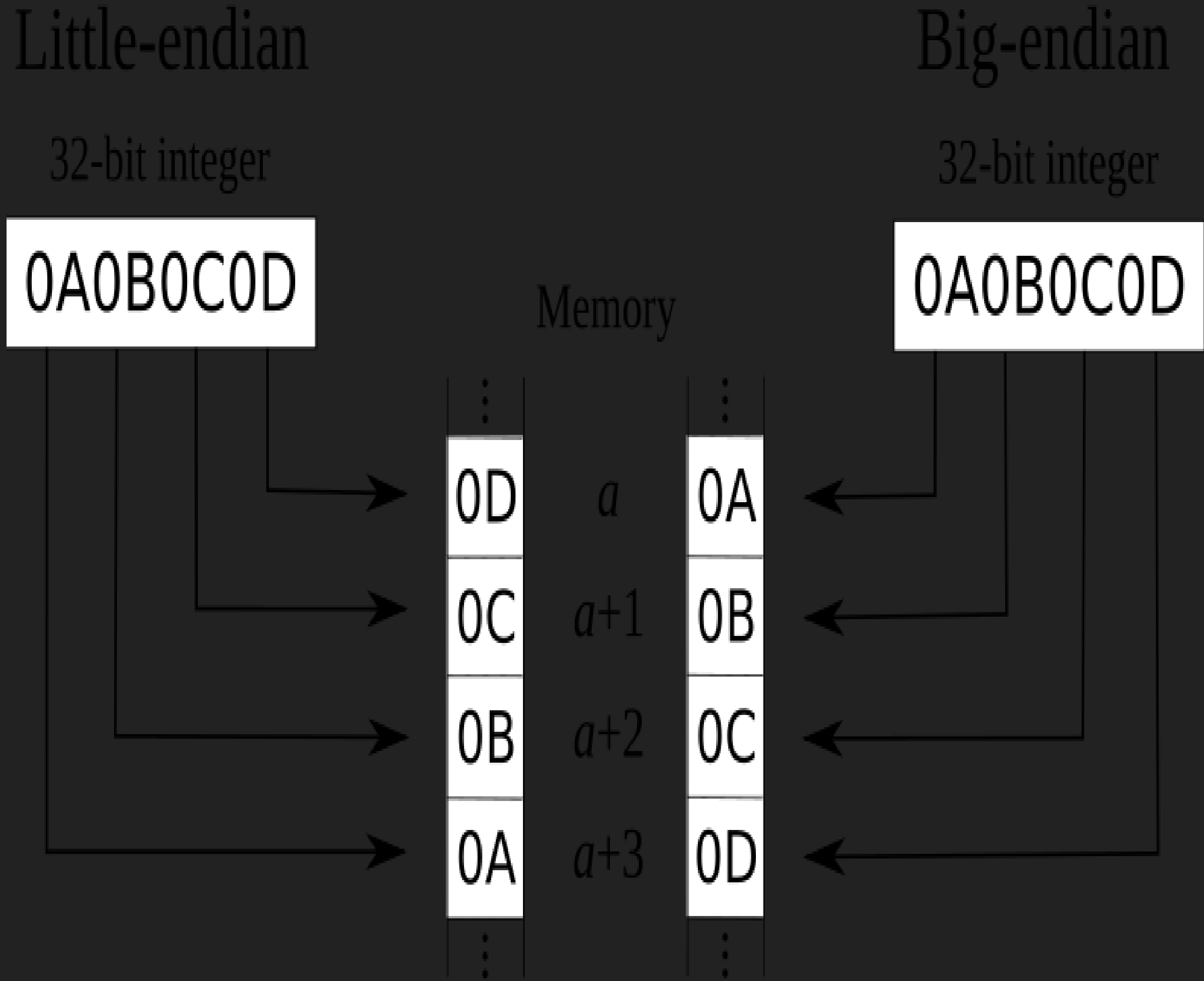
- Heap
    - Memory that is ~~dynamically allocated~~
    - Can only put things on the heap using malloc()
    - MUST be freed!!!
    - Grows bottom-up
- Stack
    - Memory that is “automatically” allocated and "freed" by the system
    - Grows top-down


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Endianness

Endianness

- Big endian: most significant byte @ smallest address
- Little endian: most significant byte @ largest address
- Base line: if you store something into memory, you should be able to read out the same value! (double flipping)



Thank you!

Feedback