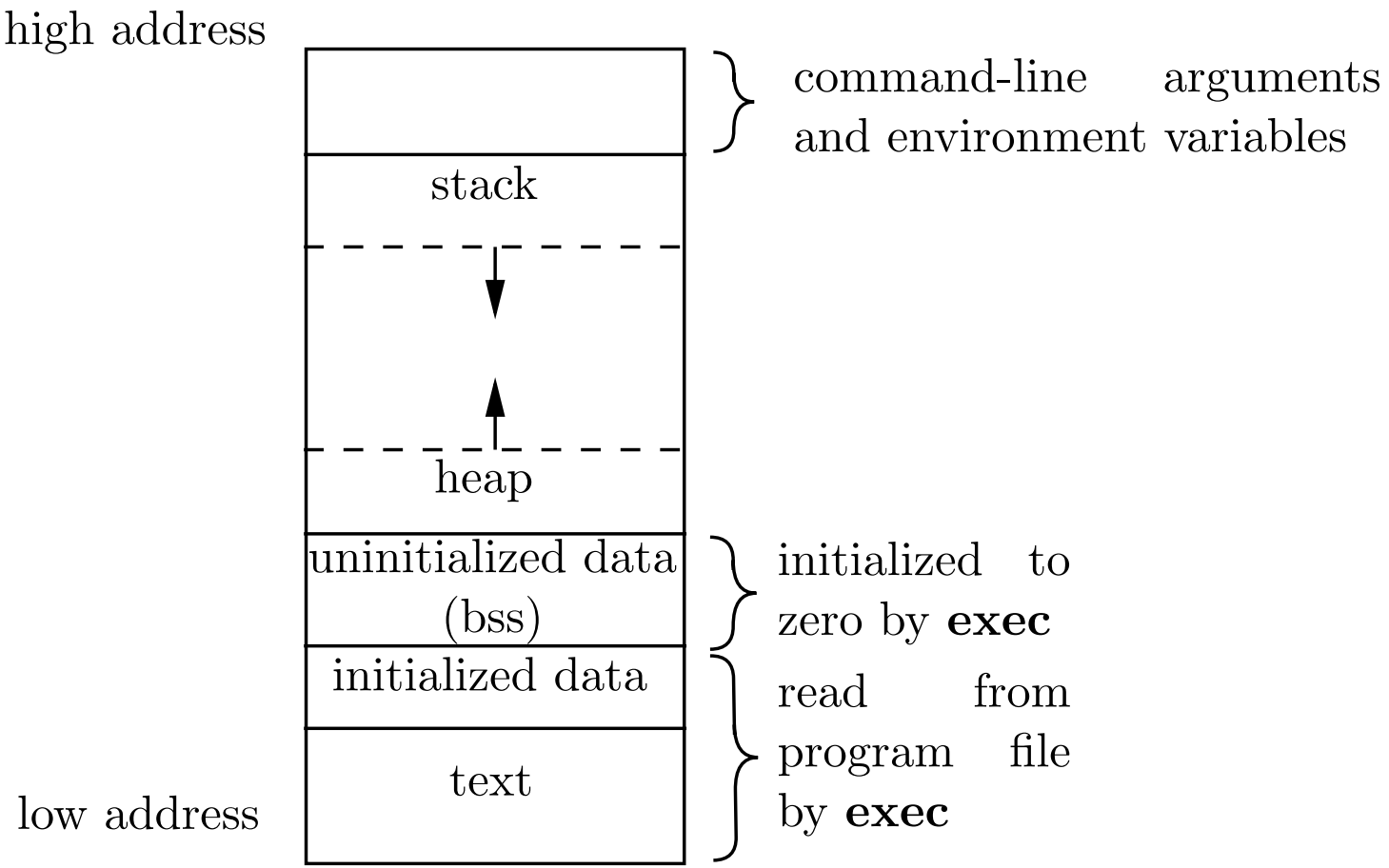
Lab 7: Memory Management

Muwaffaq Imam, Artem Kruglov,
Nikita Lozhnikov, Giancarlo Succi, Xavier Vasquez
Herman Tarasau, Firas Jolha

Innopolis University
Course of Operating Systems

29th September 2021

Memory Layout of C Program



Text or Code Segment

- Contains machine code of the compiled program. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified

Initialized Data or Data Segment

- Stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand

Uninitialized Data or .bss Segment

- Stores all uninitialized global, static, and external variables (declared with extern keyword)

Stack Segment

- Stores all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over

Heap Segment

- Part of RAM where dynamically allocated variables are stored. In C language dynamic memory allocation is done by using malloc and calloc functions

Exercise 1

- Use **size** shell command to determine the size of text, data and bss segments of any of your programs. Save the output to file ex1.txt

Pointers revision (1/3)

- Each variable represents an address in memory and a value.
- Address: &variable = address of variable
- A pointer is a variable that "points" to the block of memory that a variable represent

Pointers revision (2/3)

- Declaration: *data_type *pointer_name;*
- Example:

```
char x = 'a';
char *ptr = &x; // ptr points to a char x
```

- Pointers are integer variables themselves, so can have pointer to pointers:

```
char **ptr;
```

Pointers revision (3/3)

- Dereferencing = Using Addresses

```c
int x = 5;
int *ptr = &x;
// Access x via ptr, and changes it to 6
*ptr = 6;
// Will print 6 now
printf("%d", x);
```

Why use pointers?

- Pass-by-reference rather than value

```
void sample_func(char* str_input);
```

- Manipulate memory effectively
- Useful for arrays (Array in C - a pointer and a length)

malloc()

- void *malloc(size_t size)
  Example:

```
int array[10]; // the same as
int *array = malloc(10*sizeof(int));
```

- malloc() does not initialize the array; this means that **the array may contain random or unexpected values**

calloc()

- void *calloc(size_t nmemb, size_t size);
- The calloc() function allocates space for an array of items and **initializes the memory to zeros**

realloc()

- void *realloc(void *ptr, size_t size);
- The realloc() function changes the size of the object **pointed to by *ptr*** to the **size specified by *size***

free()

- void free(void *ptr)
- Releases memory allocated by malloc(), calloc() or realloc()

```
int *myStuff = malloc(20 * sizeof(int));
if (myStuff != NULL)
    {
        /* more statements here */
        /* time to release myStuff */ free( myStuff );
    }
```

Exercise 2

- Write a C program that dynamically allocates memory for an array of N integers, fills the array with incremental values starting from 0, prints the array and deallocates the memory. Program should prompt the user to enter N before allocating the memory.

Exercise 3

- Complete the following code template according to the comments. The purpose of the program is to create an initial array of a user-specified size, then dynamically resize the array to a new user-specified size.

Exercise 4

- Write your own realloc() function using malloc() and free()
  - realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.
  - Newly allocated memory will be uninitialized
  - If ptr is NULL, the call is equivalent to malloc(size)
  - If size is equal to zero, the call is equivalent to free(ptr)
  - Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc()

Exercise 5

- Find and fix all the code that generates segmentation faults

```
#include <stdio.h>
int main() {
    char **s;
    char foo[] = "Hello World";
    *s = foo;
    printf("s is %s\n",s);
    s[0] = foo;
    printf("s[0] is %s\n",s[0]);
    return(0);
}
```

End of lab 7