**Lab: Understanding Pointers, Invalid Pointers, and Memory Faults**

**Objective:**

This lab will help you understand:

* Definitions and initialization of pointers.
* Invalid pointers and segmentation faults.
* The difference between segmentation faults and page faults.
* Proper pointer initialization and the significance of NULL pointers.

**Lab 1: Definitions and Basic Pointer Usage**

**Concept:**

* A pointer is a variable that stores the **memory address** of another variable.

**Task:**

* Declare an integer variable.
* Create a pointer to store its address.
* Print the address and value using the pointer.

**Code:**

c

#include <stdio.h>

int main() {

int num = 42; // Integer variable

int \*ptr = &num; // Pointer storing the address of num

printf("Value of num: %d\n", num);

printf("Address of num: %p\n", &num);

printf("Value stored at ptr: %p\n", ptr);

printf("Value pointed by ptr: %d\n", \*ptr);

return 0;

}

**Expected Output:**

* Prints the integer value, its memory address, and the dereferenced pointer value.

**Lab 2: Invalid Pointers and Segmentation Faults**

**Concept:**

* An **invalid pointer** is a pointer that points to an unknown or unallocated memory location.
* Accessing such memory **causes a segmentation fault**.

**Task:**

* Declare an uninitialized pointer.
* Try to access it and observe the segmentation fault.

**Code (Will Cause Segmentation Fault):**

c

#include <stdio.h>

int main() {

int \*ptr; // ❌ Uninitialized pointer

printf("Value at ptr: %d\n", \*ptr); // Accessing uninitialized memory (SEGFAULT)

return 0;

}

**Expected Outcome:**

* A segmentation fault occurs due to accessing an uninitialized pointer.

**Lab 3: Segmentation Fault vs Page Fault**

**Concept:**

* **Segmentation Fault**: Accessing an invalid memory region.
* **Page Fault**: The OS brings a page from disk to RAM when required.

**Task:**

* Attempt to write to a **read-only** string (segfault).
* Dynamically allocate memory, access it, and observe page behavior.

**Code (Segmentation Fault Example)**

c

#include <stdio.h>

int main() {

char \*str = "Hello"; // Read-only memory

str[0] = 'h'; // ❌ Attempting to modify (SEGFAULT)

return 0;

}

**Expected Outcome:**

* **Segmentation fault** due to modifying read-only memory.

**Code (Page Fault Example - Works Fine)**

c

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr = (int \*)malloc(1000000 \* sizeof(int)); // Large memory allocation (Page fault handled by OS)

for (int i = 0; i < 1000000; i++) {

arr[i] = i; // Accessing memory triggers page allocation

}

printf("Memory allocated and used successfully.\n");

free(arr); // Free allocated memory

return 0;

}

**Expected Output:**

* Memory is dynamically allocated and accessed without errors.
* The OS **handles page faults automatically** when required.

**Lab 4: Pointer Initialization & NULL Pointer Significance**

**Concept:**

* **Uninitialized pointers** can point to random locations, causing crashes.
* **NULL pointers** are used for safe pointer checks.

**Task:**

* Initialize a pointer with NULL.
* Check before dereferencing.

**Code:**

c

#include <stdio.h>

int main() {

int \*ptr = NULL; // Proper initialization

if (ptr == NULL) {

printf("Pointer is NULL. Safe to proceed.\n");

} else {

printf("Value at ptr: %d\n", \*ptr);

}

return 0;

}

**Expected Output:**

* The program prints "Pointer is NULL. Safe to proceed.", preventing crashes.

**Lab Questions**

1. What is a segmentation fault, and how does it occur?
2. What is a page fault? How does the OS handle it?
3. Why should you always initialize pointers?
4. How does using NULL pointers prevent errors?
5. What happens when you try to modify a string literal?