1. Complete Lecture: Non-Linear Flow of Control (ASCII)

I. The Foundation: Reviewing the Call Stack

Standard program execution is **linear** and **synchronous**. When a function is called, the system manages memory using the call stack.

- **The Call Frame:** Every pending function call creates a memory area called a "call frame" or "stack frame."
- **Contents:** A frame holds local variables, arguments, and most critically, the **return address** (the instruction pointer where execution should resume).
- **Linearity:** The stack grows predictably (e.g., `main` calls `A`, `A` calls `B`). Execution always returns to the function directly below it on the stack (LIFO).

II. Example 1: Signal Handlers (Asynchronous Events)

Signal handlers allow the Operating System (OS) to interrupt the program at any moment to communicate an event (e.g., divide-by-zero, timer expiry, user interrupt).

- **Trigger:** An **asynchronous** event originating from the OS kernel or hardware.
- **Mechanism:**

1. The CPU is paused immediately.
2. The OS forces the creation of a **new call frame** on top of the existing stack structure.
3. This frame belongs to the registered **Signal Handler function**.

- **Non-Linearity:** This is a non-linear flow because execution jumps from any arbitrary instruction to the handler function, without a corresponding `call` instruction in the program's source code.

III. Example 2: Context Switching (`getcontext` and `setcontext`)

This mechanism allows user-level code to freeze the entire state of the CPU and later restore it, enabling programmatic jumps between arbitrary points in execution.

What is a "Context"?

A "context" (represented by `ucontext_t` in C) is a data structure containing the snapshot of the CPU's state at a single moment. It includes:

- **Program Counter (PC) / Instruction Pointer:** The address of the next instruction.
- **Stack Pointer (SP):** The current top of the stack.
- **Registers:** The values stored in all General Purpose Registers.

The Flow: Emulating Exceptions

Using `setcontext` is key to implementing exceptions in low-level runtime environments.

Function	Action	Effect on Flow
`getcontext()`	**Save:** Saves the current CPU state (registers, PC, SP) to a context variable. (The "Try" block checkpoint).	Execution continues linearly.
`setcontext()`	**Restore:** Loads a previously saved context back into the CPU.	Execution **teleports** back to the instruction immediately following the original `getcontext()` call.

C Code Example: Context-Based Exception

```c
#include <stdio.h>
#include <ucontext.h>
#include <signal.h>

static ucontext_t checkpoint_context;
volatile int exception_caught = 0;

void exception_handler(int sig) {
    printf("\n[Signal Handler] Caught SIGFPE.\n");
    // Non-Local Return: Instantly jump back to the saved context.
    exception_caught = 1; // Set flag to exit the "if" block
    setcontext(&checkpoint_context);
}

int main() {
    signal(SIGFPE, exception_handler);
```

```c
    // Save the "Try" context (the checkpoint)
    getcontext(&checkpoint_context);

    if (exception_caught == 0) {
        printf("[Main] Context saved. Attempting illegal operation...\n");
        int a = 10;
        int b = 0;
        int result = a / b; // Causes SIGFPE signal
        printf("[Main] Result: %d\n", result);
    }
    else {
        // Execution jumps here from setcontext
        printf("[Main] Recovered! Flow restored to the checkpoint.\n");
    }

    return 0;
}
```

IV. Example 3: Constructor Functions (Pre-execution Initialization)

This non-linear flow occurs before the program's explicit starting point (`main`).

- **The Rule:** Any global object or variable initialized by a constructor function **must** execute *before* the `main` function starts.
- **Mechanism:** The **Dynamic Linker** handles this flow by executing the constructors in all loaded libraries before jumping to the program's main entry point.

Forcing Non-Linear Flow with `LD_PRELOAD`

Using the environment variable `LD_PRELOAD`, we force the Dynamic Linker to load a custom shared library (`mylib.so`) and execute its constructor before `main`.

C Code Example: Constructor Injection

`hack.c` (The Shared Library)

```c
// hack.c
#include <stdio.h>

// The compiler attribute that marks this function for pre-execution.
```

```c
__attribute__((constructor))
void my_init_hook() {
    printf("\n[Constructor Hook] --> Code running BEFORE main() starts!\n");
}
```

main.c (The Target Program)

```c
// main.c
#include <stdio.h>
int main() {
    printf("[Main] --> Now executing the main function.\n");
    return 0;
}
```

Execution Guide

```bash
# 1. Compile the library and the program
gcc -shared -fPIC -o libhack.so hack.c
gcc -o normal_program main.c

# 2. Run with the non-linear flow enforced
LD_PRELOAD=./libhack.so ./normal_program
```

Output:

```
[Constructor Hook] --> Code running BEFORE main() starts!
[Main] --> Now executing the main function.
```

V. Summary Table (ASCII)

Mechanism	Trigger	Stack Behavior	Flow Type
Signal Handler	Asynchronous OS Event	Pushes new frame on top	Interrupt
setcontext	Explicit Function Call	Overwrites CPU state (Non-local jump)	Jump/Restore
Constructors	Program Load / Linker	Execution occurs before the defined entry point (`main`).	Pre-execution