INTERMEZZO A: Non-Linear Flow of Control

Gene Cooperman

INTERMEZZO: Non-Linear Flow of Control

Objective: To understand how program execution flow can deviate from the standard sequential path using three primary mechanisms: Signal Handling, Context Switching, and Pre-execution Initialization.

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 (a signal sent by the operating system), such as a divide-by-zero, timer expiry, or user interrupt.

- **Trigger:** An **asynchronous** event originating from the OS kernel or hardware.
- **Comparison to Exceptions:** A signal handler processes this asynchronous event in a similar philosophical manner to **exception handlers** seen in higher-level languages like Java and C++. When an asynchronous fault occurs, the signal handler is invoked, much like a `catch` block is invoked when an exception is thrown.
- **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 is the current state of the CPU, and especially the current state of the registers, including the **Program Counter (Instruction Pointer)** and **Stack Pointer**.

The Flow: Implementing Try-Catch using Signals

The `try-catch` syntax for exception handlers in Java or C++ can be implemented as a signal handler combined with a call to `setcontext`.

1. **The "Try" Block:** At the beginning of a high-level `try` block, the runtime performs a call to `getcontext()`. This saves the current, stable CPU state into a context variable (the "checkpoint").
2. **The Fault/Exception:** If code inside the `try` block triggers a fatal error, the OS sends a **signal**.
3. **The Signal Handler (The "Catch"):** The pre-registered **signal handler** function is executed.
4. **The Unwind:** Inside the signal handler, the non-linear action occurs: a call to **`setcontext()`** using the checkpoint saved in step 1.

- **Effect:** This instantly **unwinds the stack** (discarding the signal handler's frame and the frames that led to the fault), and execution resumes at the exact instruction immediately following the original `getcontext()` call, thus returning to an earlier call frame on the stack.

Function	Action in a Try/Catch implementation	Effect on Flow
`getcontext()`	**Save:** Executed at the start of the `try` block.	Execution continues linearly.
Signal Handler	**Interrupt:** Invoked on fault (e.g., divide by zero).	Interrupts linear flow, pushes new frame.
`setcontext()`	**Restore:** Called inside the signal handler to jump back to the `getcontext` location.	Execution **teleports** (non-local jump) back to the saved state.

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

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

- **The Rule:** If a global object or variable is initialized by a **constructor function**, this must occur even **before the normal execution of the "main" function**.
- **Mechanism:** The **Dynamic Linker** (the program loader) 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`

This scenario can be forced by, for example, setting the environment variable, `LD_PRELOAD=/full_path/mylib.so` before executing a program.

- If `mylib.so` has a global variable or object initialized by a constructor function, then the constructor function will execute even before the function `main`, due to the use of `LD_PRELOAD`.

Flow of Control Summary

The flow is dramatically altered from the expected: **Loader (Linker) Library Constructor Main function**.

V. Summary Table

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

---

VI. Going Deeper

Mechanism	man page
Signal Handler	`man 2 signal` (register a signal handler function); `man 2 kill` (send signal)
setcontext	`man 2 setcontext` (includes setcontext and getcontext)
Loader/linker	`man ld.so` (and search on `LD_PRELOAD`)
Constructors (used with `LD_PRELOAD`)	Java: built into spec for classes C: add `__attribute__((constructor))` after return type of the function

Example Code:

See `A-INTERMEZZO-2-C-code-setcontext.pdf` and `A-INTERMEZZO-3-C-code-LD_PRELOAD.pdf`, in this directory, for example C code.