INTERMEZZO A: C Code Example: Context Switching for Exception Emulation

Gene Cooperman

C Code Example: Context Switching for Exception Emulation

This example demonstrates how to combine **Signal Handling** (Example 1) with **`setcontext`** (Example 2) to achieve a non-local jump that simulates a high-level exception (`try/catch` block) by "rewinding" the program flow.

File: `context_exception.c`

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

// The "checkpoint" variable to store the CPU state (registers, PC, stack pointer)
static ucontext_t checkpoint_context;

// A flag to simulate the logic flow of a try/catch block
// 'volatile' ensures the compiler doesn't optimize away checks on this variable.
volatile int exception_caught = 0;

// This is our Signal Handler (the low-level "catch" mechanism)
void exception_handler(int sig) {
    printf("\n[Signal Handler] Caught signal %d (SIGFPE).\n", sig);
    printf("[Signal Handler] Resolving error and restoring context...\n");

    // 1. Update the state to indicate we handled the error
    exception_caught = 1;

    // 2. Non-Local Return: JUMP
    // We overwrite the current CPU state with the state saved earlier.
    // Execution will immediately resume at the point where getcontext was called.
    setcontext(&checkpoint_context);
    // The program will NEVER reach code immediately after setcontext().
}

int main() {
    // Register the signal handler for SIGFPE (Floating Point Exception / div by zero)
    signal(SIGFPE, exception_handler);

    printf("[Main] Starting program.\n");

    // SAVE THE CONTEXT (The "Try" block entry point)
    // getcontext saves the current registers and stack ptr into 'checkpoint_context'
    // Execution proceeds immediately after this line (getcontext returns 0).
    getcontext(&checkpoint_context);

    if (exception_caught == 0) {
        // --- FIRST PASS (Linear Flow) ---
        printf("[Main] Context saved. About to attempt dangerous operation...\n");

        // --- DANGER ZONE ---
        // This causes an immediate interrupt/signal (SIGFPE)
        int a = 10;
        int b = 0;
        int result = a / b;

        // This line is unreachable in the first pass
        printf("[Main] Result: %d\n", result);
    }
    else {
        // --- SECOND PASS (Non-Linear Flow) ---
        // Execution jumps directly here because setcontext restored the IP
        // to the getcontext line, but the variable 'exception_caught' is now 1.
        printf("[Main] Recovered! Execution resumed at the checkpoint.\n");
    }

    printf("[Main] Program finishing normally.\n");
    return 0;
}
```

Lecture Talking Points

1. **The Checkpoint (`getcontext`)**:

- This function saves the entire CPU state (including the instruction pointer pointing to the line *after* `getcontext`) into `checkpoint_context`. This is the **linear flow**.

2. **The Interruption (Signal)**:

- The division by zero (`a / b`) causes the OS to stop the program, push a frame for `exception_handler` onto the stack, and execute the handler. This is the **asynchronous, stack-building** phase.

3. **The Teleport (`setcontext`)**:

- Inside the handler, `setcontext(&checkpoint_context)` is the critical non-linear operation.

- It **overwrites** the CPU's registers with the saved state.
- This causes execution to immediately jump back to the `getcontext` call site, but this time, the `exception_caught` flag is set to handle the jump correctly. The stack is effectively **unwound** without the standard `ret` instruction.