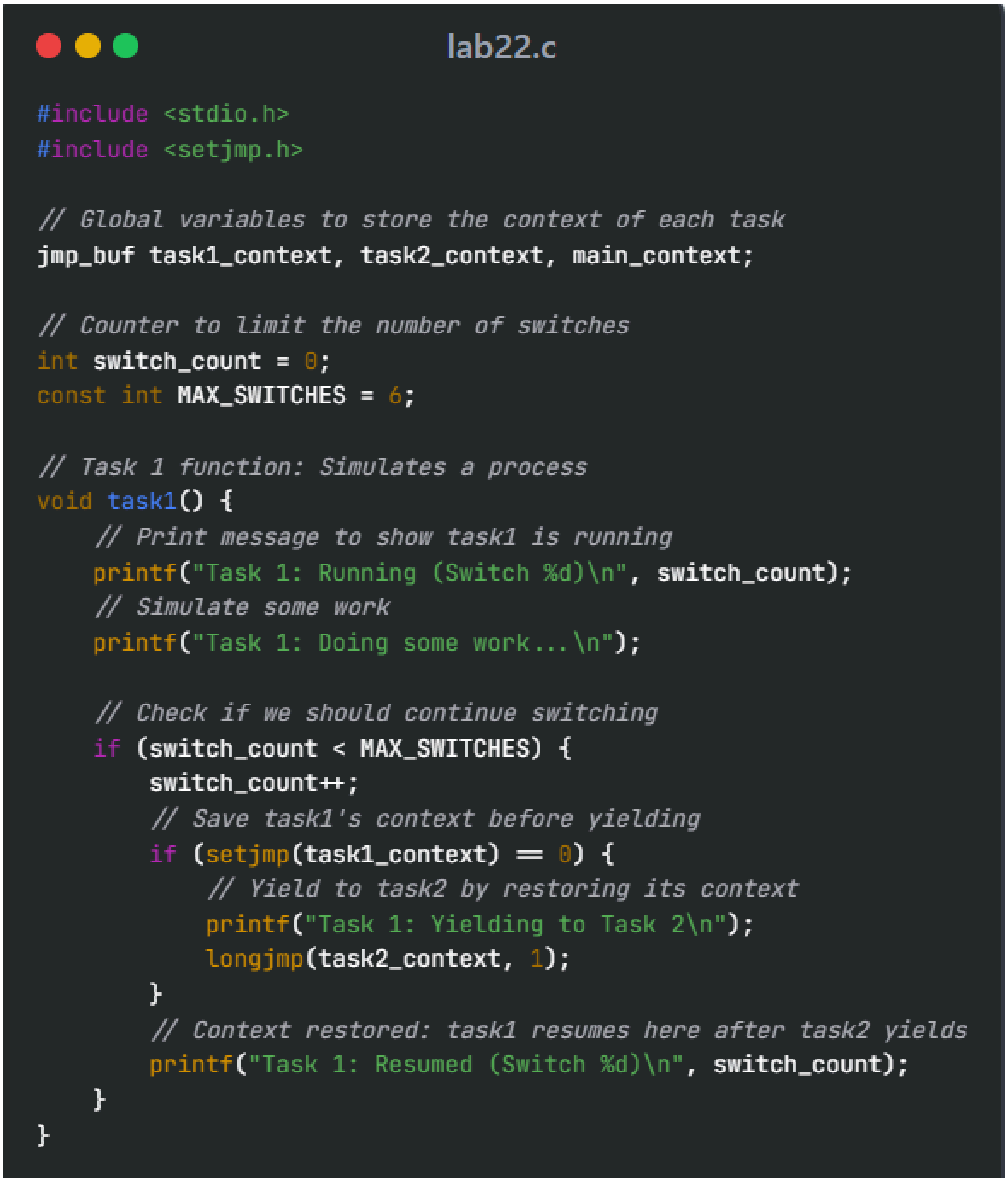
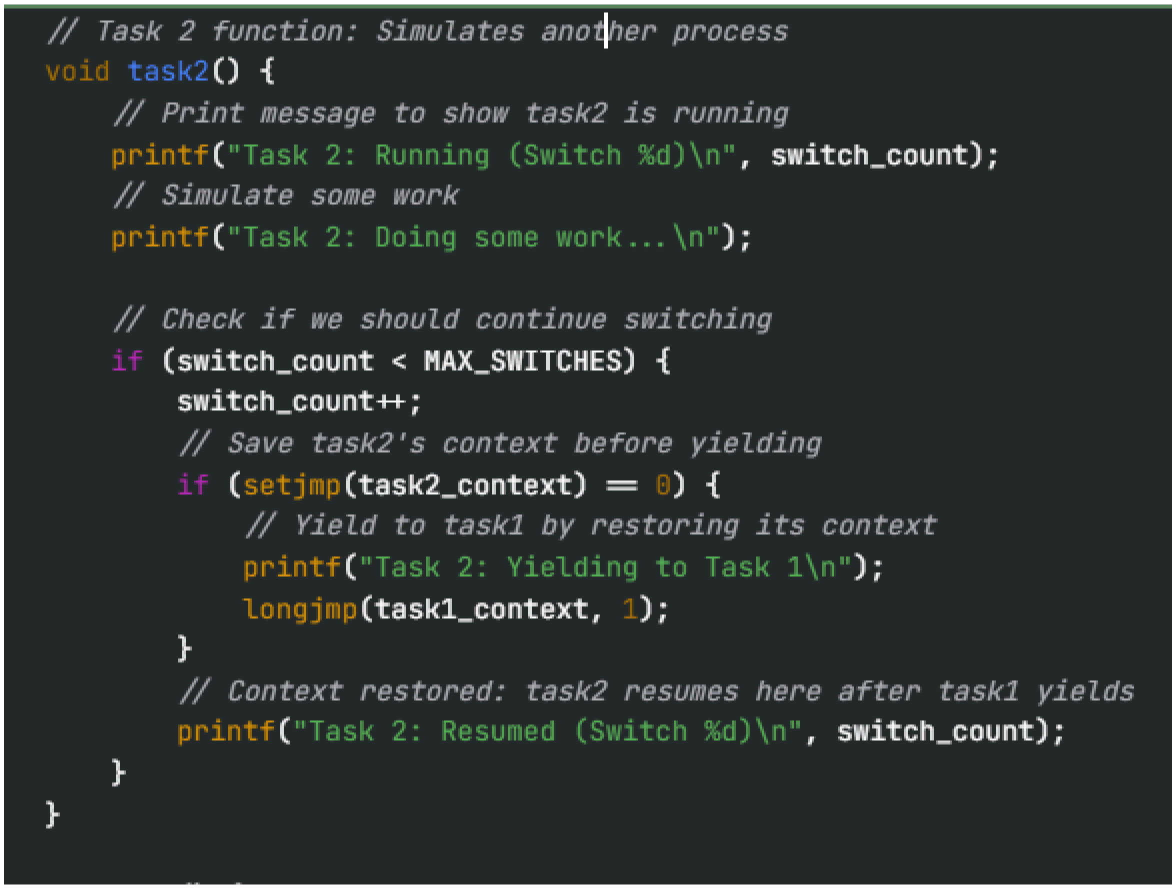
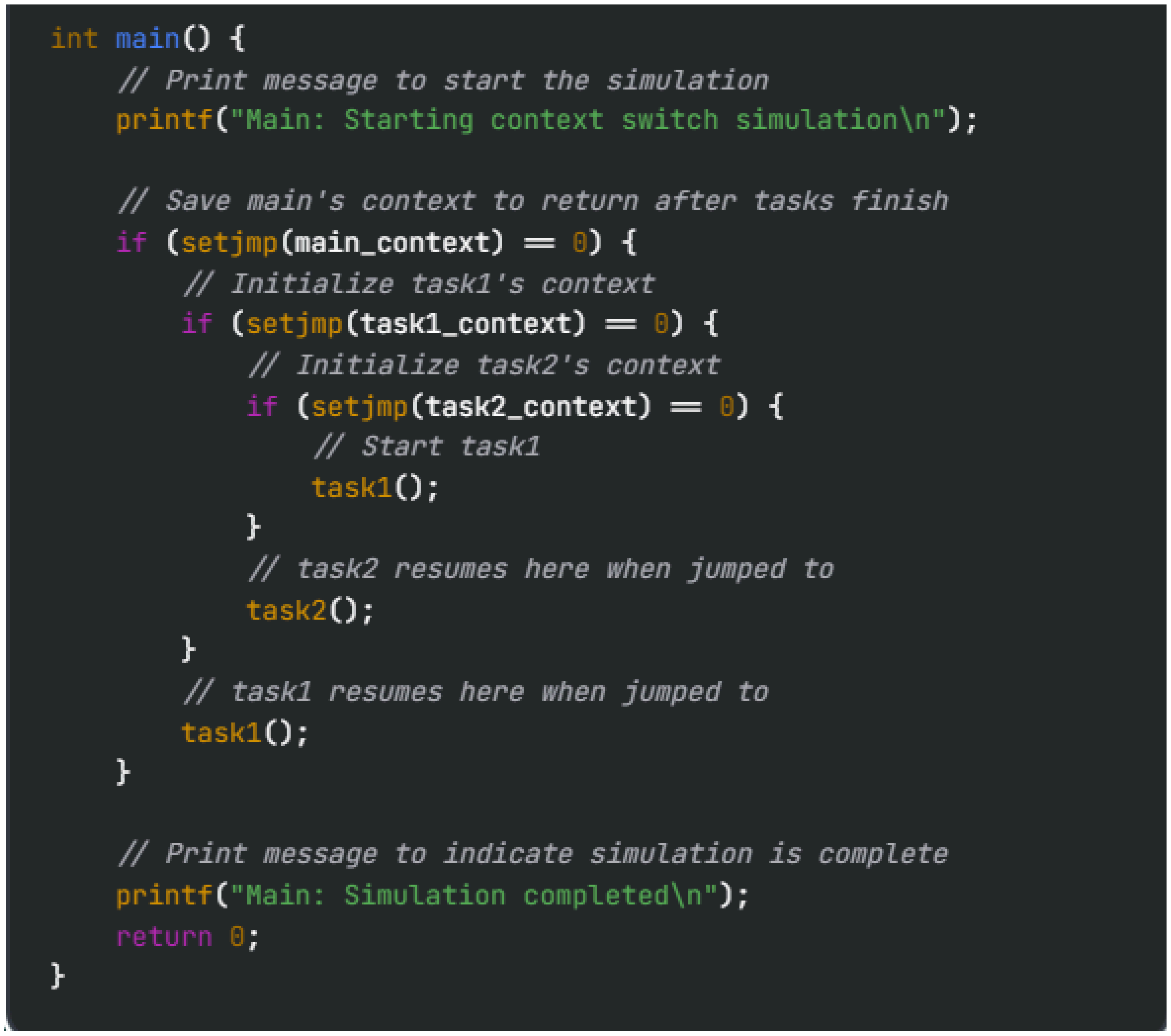


lab22.c

```c
#include <stdio.h>
#include <setjmp.h>

// Global variables to store the context of each task
jmp_buf task1_context, task2_context, main_context;

// Counter to limit the number of switches
int switch_count = 0;
const int MAX_SWITCHES = 6;

// Task 1 function: Simulates a process
void task1() {
    // Print message to show task1 is running
    printf("Task 1: Running (Switch %d)\n", switch_count);
    // Simulate some work
    printf("Task 1: Doing some work...\n");

    // Check if we should continue switching
    if (switch_count < MAX_SWITCHES) {
        switch_count++;
        // Save task1's context before yielding
        if (setjmp(task1_context) == 0) {
            // Yield to task2 by restoring its context
            printf("Task 1: Yielding to Task 2\n");
            longjmp(task2_context, 1);
        }
        // Context restored: task1 resumes here after task2 yields
        printf("Task 1: Resumed (Switch %d)\n", switch_count);
    }
}
```



```c
// Task 2 function: Simulates another process
void task2() {
    // Print message to show task2 is running
    printf("Task 2: Running (Switch %d)\n", switch_count);
    // Simulate some work
    printf("Task 2: Doing some work...\n");

    // Check if we should continue switching
    if (switch_count < MAX_SWITCHES) {
        switch_count++;
        // Save task2's context before yielding
        if (setjmp(task2_context) == 0) {
            // Yield to task1 by restoring its context
            printf("Task 2: Yielding to Task 1\n");
            longjmp(task1_context, 1);
        }
        // Context restored: task2 resumes here after task1 yields
        printf("Task 2: Resumed (Switch %d)\n", switch_count);
    }
}
```



```c
int main() {
    // Print message to start the simulation
    printf("Main: Starting context switch simulation\n");

    // Save main's context to return after tasks finish
    if (setjmp(main_context) == 0) {
        // Initialize task1's context
        if (setjmp(task1_context) == 0) {
            // Initialize task2's context
            if (setjmp(task2_context) == 0) {
                // Start task1
                task1();
            }
            // task2 resumes here when jumped to
            task2();
        }
        // task1 resumes here when jumped to
        task1();
    }

    // Print message to indicate simulation is complete
    printf("Main: Simulation completed\n");
    return 0;
}
```

Context Switching Simulation – Detailed Report

Step 1: Context Initialization in Main

At the beginning of the main() function, we display "Main: Starting context switch simulation" to indicate the simulation has started. Right after, the program saves the current execution state using setjmp(main_context). Initially, setjmp() returns zero, allowing the program to continue. If later a longjmp() returns here, it will resume execution exactly from this point. This setup ensures that the main() function can regain control after switching between tasks.

Performance Note: Saving context with setjmp() is lightweight but depends slightly on the system's architecture and what variables need preserving. Although the impact is minor here, in more complex systems the cost could add up.

Step 2: Preparing Contexts for Tasks

After securing the main context, the simulation captures the states for two tasks by calling setjmp(task1_context) and setjmp(task2_context). These prepare save points for task1() and task2() respectively. If longjmp() is invoked, the program can return directly to the saved position inside each task. These setjmp calls are made early, so when switching later, each task resumes from a well-defined state.

Performance Note: For two tasks, the overhead is minimal. But as task count increases, or if each context contains large local data, the time taken for each setjmp grows accordingly.

Step 3: Task Execution and Switching Between Them

task1() starts by performing some basic output operations. After doing some work, it saves its state with setjmp(task1_context) and then immediately switches to task2() using longjmp(task2_context, 1). Inside task2(), the same pattern happens: it prints its messages, saves its state with setjmp(task2_context), and jumps back to task1(). This back-and-forth context switching simulates the cooperative multitasking behavior typical in early systems.

Performance Note: Although longjmp() is quick, every context switch involves reloading CPU registers and stack pointers, which could slow down performance if switches happen extremely often or if many contexts are involved.

Step 4: Resuming Tasks and Repeating the Switch

When task2() hands control back, task1() resumes execution precisely after the last setjmp(). It prints a "resumed" message indicating the switch number. Then it continues doing work and yields again to task2(). This cycle repeats until the maximum allowed switches (MAX_SWITCHES) are reached. The simulation shows how tasks can alternate without true threading by saving and restoring execution states.

Performance Note: For small workloads, the performance hit is barely noticeable. However, in more intensive cases or with multiple switches happening rapidly, the cumulative cost might lead to lag, especially if task stacks are large.

Step 5: Wrapping Up the Simulation

Once the switching cycle finishes, the tasks return control back to main(). Finally, the program prints "Main: Simulation completed," signaling that no further context switching will occur. This last part runs after all tasks have completed their switching duties.

Performance Note: The final section carries no switching overhead and runs normally like any sequential C program would.

Challenges and Performance Discussion

1. Context Switch Cost:
Every setjmp/longjmp introduces minor computational effort. While small for two tasks, managing 10 or 100 tasks with frequent switches would make this overhead more significant, affecting responsiveness.

2. Scalability Issues:
This simulation is hardcoded for two tasks. Expanding it would require maintaining many separate contexts, increasing complexity and memory usage. Without better scheduling mechanisms, scaling becomes inefficient.

3. Memory and Stack Risks:
Saving stack states repeatedly can consume a large amount of memory, especially if each task allocates heavy local data. On systems with limited stack sizes, this could cause overflows.

4. Lack of Preemptive Switching:
Unlike modern operating systems that use interrupts to preempt tasks, this simulation depends on manual yielding (longjmp). If a task refuses to yield, it can hog execution time, creating unfair execution.

5. Debugging Difficulty:
Since program flow jumps non-linearly with longjmp, understanding the current state or call stack while debugging becomes much harder. Tracking bugs or unexpected behavior across context switches requires more effort.

Final Thoughts

This simulation provides a clean, manual example of task switching in C using setjmp and longjmp. It illustrates fundamental ideas but would need significant redesigns for larger or real-world systems. While it teaches the importance of saving and restoring state, the technique is better suited for learning and simple cooperative multitasking than for real scalable applications.