Larry Lu, Andrew Choi

20 March 2023

ECE 3140

Lab 3: Concurrency

Design

In this lab, we gained experience in implementing concurrency, particularly using the FRDM-KL46Z board. The provided code reflects our efforts to manage multiple processes using a custom concurrency management system. Our goal was to create a robust and efficient solution to handle concurrent processes, enabling seamless execution and process switching.

Our high-level architecture consists of several decoupled components: a utilities file [utils.c] for manipulating the built-in LEDs on the FRDM-KL46Z board; a concurrency library [concur.c] providing essential functions for allocating and deallocating stack space for processes, starting processes, and other related operations; a processing module [process.c] for creating, starting, and selecting processes; and test files [test_1.c, test_2.c] for visualizing our concurrency implementation.

The process.c module, as shown in the provided code, captures the process state through a struct (struct process_state) containing information such as pointers to the current and original stack pointers, a pointer to the next process, and the stack size. We

employ a linked list data structure to manage multiple processes, with each process represented as a node. This enables us to add processes to or remove them from a queue as needed.

To ensure smooth execution and process management, we carefully designed our system to handle various scenarios such as processes running for different durations and processes with different priorities. This required a meticulous approach when implementing core functions and managing the process queue.

Implementation

In process.c, we implement several core functions responsible for creating, starting, and selecting processes.

process_create(void (*f)(void), int n): When creating a process, we utilize the function process_stack_init to allocate stack space for a stack pointer. We then allocate memory for the process itself and add it to the queue using the enqueue function. This function ensures that the newly created process is properly initialized and added to the queue for execution.

timer_setup(void): This function sets up the PIT (Programmable Interrupt Timer) to trigger interrupts for process switching. It configures the timer to generate an interrupt at regular intervals, allowing the system to switch between processes and execute them concurrently.

process_start(void): This function enables interrupts, configures the PIT using timer_setup, and starts the first process in the queue. It sets the stage for concurrent

execution by initializing the necessary hardware components and starting the execution of processes.

process_select(unsigned int * cursp): During process selection, we deallocate stack space from the current process (if it's finished) and set the current process to the next process in the queue. This function is responsible for managing the process queue, ensuring that processes are executed in the correct order and that completed processes are properly deallocated.

Additionally, the provided code includes two helper functions, enqueue(process_t* new_process) and dequeue(void), which manage the process queue by adding new processes to the end of the queue and removing processes from the front of the queue, respectively. These functions are essential for maintaining the order of process execution and ensuring that the queue remains well-organized.

Code Review

We were provided with a test file [test_0] that examines a simple scenario, in which we create two processes: one toggles the red LED 6 times, and the other toggles the green LED 13 times. We verified our implementations by loading the program onto our FRDM-KL46Z board and manually checking if the red LED toggled 6 times and the green LED toggled 13 times.

Additionally, we created two more test files [test_1, test_2] that evaluate two other situations. In [test_1], we designed a simple scenario in which there is only one process and one call to process_create(). In this situation, the first process should be performed

as normal until completion, at a regular speed because there is no second process to slow it down. In [test_2], we consider the edge case where one of the processes continues forever by recursively calling process_create() on itself. Process 1 blinks the red LED for a limited time and process 2 blinks the green LED forever by adding itself back to the process queue upon termination. Both test cases ran to completion, and the intended behavior was observed.