Lab 3 Concurrency

Deadlines & Grading

- **Prelab: (5pts)** Fri March 10 at **11:59 PM** on Canvas only one team member needs to submit.
- **Code and report submission**: (90pts)Friday, March 17 at **11:59 PM** on CMS, **groups of 2**
- **Team Evaluation: (5pts)** Tues, March 21st **11:59 PM** on Canvas, *individual*
- **Grading**: 10% of total grade

Prelab Questions (5pts)

In order to complete lab assignments successfully and on time, it is best to start work early. To encourage this and to help you clarify your tasks and clear up any lingering questions about the material, this lab has a pre-lab that is due a week before the actual submission. This lab is more complex than the previous two labs. In order to prepare, and discover some of the pitfalls, try to start the conversation with your new lab partner by explaining the following points to each-other. Only one of you needs to fill out the pre-lab.

Step 1: Please read the entire handout! This is the time to clarify anything with the course staff.

Step 2: Please answer the following questions in 1-3 sentences each. Use your own words. You may want to type out the answers in a separate document and then cut/paste them into the submission form.

Question 1: Were you able to make contact and meet with your lab partner? Hopefully this is a yes. If not what did you try to make this happen?

Question 2: Read through the support code. For the assembly code, it might make sense to use a high-lighter and color code sections of the assembly code on how/when they get called. Describe what happens when a context switch occurs?

Question 3: It's never too early to think about testing! Describe how you will test the functionality of both the linked list and processes that might be interesting/challenging as corner cases to show the functionality of the scheduler. How will you decide on the stack size for your processes?

Section I: Overview

Goal. The purpose of this lab is to give you experience with concurrency on the FRDM-KL46Z microcontroller. To successfully complete this lab, you will need to understand the following:

- I/O and Interrupts (Lab 2)
- Concurrency (time sharing) and its implementation

> **NOTE 1** We suggest that you first understand context switching on paper before you start writing C code. It is important that you have confidence in your implementation before you start, because it is challenging to find mistakes using standard testing methods.

Precautions.

- The micro-controller boards should be handled with care. Misuse such as incorrectly connecting the boards is likely to damage the device.
- These devices are *static sensitive*. This means that you can "zap" them with static electricity (a bigger problem in the winter months). Be very careful about handling boards that are not in their package. Your body should be at the same potential as the boards to avoid damaging them. For more information, check wikipedia.
- It is your responsibility to ensure that the boards are returned in the same condition as you received them. If you damage the boards, it is your responsibility to get a replacement.

Section II: Getting Started

Use the same setup instructions as in Lab 1 and Lab 2. Delete the automatically generated C-files and include the following provided files in your source.

3140.s – contains the assembly language definition of the timer interrupt and functions called when a process terminates.
3140_concurr.c – contains C definitions of functions for allocating stack space and initializing a process.
3140_concur.h – a header file with all function definitions listed **(including the ones that you must implement)**.
utils.c – contains helper functions for setting up and using the LEDs.
utils.h – a header file with the function definitions for the functions defined in *utils.c*.
lab3_t0.c – a test program that uses concurrency

Add the provided source files to your project by directly extracting them from the zip file and moving them to the "source" folder for your project. Alternatively, you can also extract them elsewhere on your computer, right click on the "source" folder under your project, choose Import->General->Filesystem, choose the folder that the files were extracted to, and then select the files and click Finish.

Create a new file *process.c* which will implement the functions needed for the lab. You can create a new file by right clicking *source* in the Project window and clicking on *New -> File* Dialog. You should add the following includes to the top of your file:

```
#include "3140_concur.h"
#include <stdlib.h>
#include <MKL46Z4.h>
```

The *stdlib.h* header file allows you to use the `malloc` and `free` functions. The *3140_concur.h* header file gives the declarations for the functions you will need to implement. The *MKL46Z4.h* header file allows you to interact with peripherals on the device as in Lab 2.

> **NOTE 2** Look through all the provided files to make sure you understand the code provided. The provided files should not be modified. Instead, all the functionality you will need to implement should be written in the file *process.c*.

Data Structures

Process State

We suggest that your implementation uses a data structure to maintain the process state of the following form:

```
    struct process_state {
        unsigned int * sp;
        /* the stack pointer for the process */
        ...
    };
```

In the presence of weak fairness of concurrent execution, a process may end up relinquishing the processor to allow another process to execute at any time. On the KL46Z , we will achieve this by using timer interrupts. However, it may be advisable to disable these interrupts temporarily inside a process to create larger atomic operations and ensure that it does not get switched out of the processor. The functions that support disabling/enabling the interrupts, respectively, are:

```
PIT->CHANNEL[0].TCTRL = 1;
PIT->CHANNEL[0].TCTRL = 3;
```

When a process terminates, the next process should be automatically selected for execution by calling your `process_select()` function.

Linked Lists

Linked list data structures are commonly used to implement queues and stacks. They can easily grow and shrink, and they also make useful tasks for more complex process queues, e.g. insertion, easy. A simple linked list node that holds an integer is given by:

```
struct mylist {
        int val;
        struct mylist *next;
};
```

You can use a single pointer to store the beginning of a linked list. Suppose this pointer is called `list_start`.

Initially, the list is empty. We need a special value of `list_start` that indicates this. In C, a common practice is to use `NULL` (integer 0) to indicate it. Hence, an empty list would be declared and initialized by

```
struct mylist * list_start = NULL;
```

If `elem` is a pointer to another `struct mylist` element, then we can insert it at the beginning of our list by the following operations:

```
elem->next = list_start;
list_start = elem;
```

The operation `elem->next` is shorthand for `(*elem).next`. We can traverse the list one item at a time as follows:

```
struct mylist *tmp;
   for (tmp = list_start; tmp != NULL; tmp = tmp->next) {
      // tmp->val is the integer of interest
      // do something here
   }
```

To insert an element at the end of the list is a bit more complicated. This is because there are two cases: *(i)* the list is currently empty; or *(ii)* the list has some items in it. If the list is empty, then the operation is easy. If the list is not empty, we need to traverse the list to find the last element in it, and then add the new one to the end of the list. The following code does this (assuming that `elem` is the new element to be added to the list):

```
struct mylist *tmp;
if (list_start == NULL) {
	list_start = elem;
	elem->next = NULL;
}
else {
	tmp = list_start;
	while (tmp->next != NULL) {
		// while there are more elements in the list
		tmp = tmp->next;
	}
	// now tmp is the last element in the list
	tmp->next = elem;
	elem->next = NULL;
}
```

We can remove the first element of the list in a straightforward way:

```
if (list_start == NULL) {
	elem = NULL;
}
else {
	elem = list_start;
	list_start = list_start->next;
}
```

In the above code, `elem` is the first element, and `elem->val` is the value. Normally we do not use `elem->next` at this point since we have removed it from the list. Sometimes programmers set `elem->next = NULL` just to be safe. A similar technique can be used to remove the last element of the list. While you can use this basic version of a linked list, a useful extension would be a slightly more complex list that keeps track of both the start and end, so that you can quickly insert tasks at the end without having to traverse the list firsts.

Section III: Assignment

Part 1: Implementing Concurrency

For this part of the lab assignment, you must implement three functions. These functions are directly called by users of your concurrency package and their functionality will be described below.

`int process_create (void (*f)(void), int n);`

This function creates a process that starts at function `f`, with an initial stack size of `n`. It should return `-1` on an error, and `0` on success. The implementation of this function may require that you allocate memory for a `process_t` structure, and you can use `malloc()` for this purpose.

The state of a process can be initialized by calling the provided function `process_stack_init` (you can find the header of this function in 3140_concur.h) which allocates a new stack of size at `n*4` bytes of free space in addition to an initialization context for the process. It returns the value of the stack pointer. This function returns `NULL` if the stack could not be allocated.

`void process_start (void);`

As discussed in class, a context switch occurs on a timeout that is triggered by a timer interrupt. For this lab, we will use the periodic interval timer described in Lab 2. As such, `process_start()` must:

1. Set up a period interrupt to be generated using the `PIT` interrupt. You will also need to call `NVIC_EnableIRQ` here. The interrupt handler itself is provided for you in *3140.s*, so you do NOT need to provide a C implementation of it.
2. Make sure all data structures you need are correctly initialized.
3. End with a call to `process_begin()`, which initiates the first process.

`unsigned int * process_select(unsigned int * cursp);`

This function is called by our provided (PIT0) interrupt handler code to select the next ready process. `cursp` will be the current stack pointer for the currently running process (`NULL` if there is no currently running process). If there is no process ready, this should return `NULL`; if there is a ready process, this must return the value of the stack pointer for the next ready process. Your implementation must always maintain the global variable:

```
process_t *current_process
```

as the currently running process. This variable is set to `NULL` when a process terminates, and should also be `NULL` until `process_start()` is called.

NOTE 3 You must use good programming practice and free memory that is no longer being used. This includes linked list nodes and process stacks for terminated processes. Linked list nodes can be freed using the function `free`. Process stacks can be freed using the function `process_stack_free` which is defined in *3140_concur.h* and implemented in *3140_concur.c*.

Task

Implement the three functions (`process_create`, `process_start`, and `process_select`) described above in *process.c*.

Part 2: Testing your Implementation

We have provided a test case (*test_0.c*) for your use as a basic check. Your lab should pass this test case, but it is not comprehensive enough to ensure that your code is functionally correct. You should design your own test cases in order to test the functionality.

Task

You must provide *two* additional tests to demonstrate that your implementation works correctly. Each test case should focus on some non-trivial aspect of the scheduler's behavior. Some examples are:

- What happens if there is only one call to `process_create()`?
- What if a process exits immediately?
- What if there are many short processes and one long process?

You must come up with at least one original test case (not one of the above). Your test cases should be documented (commented) to explain what they are testing along with a description of expected behavior. Name these files *test_1.c* and *test_2.c*.

Section IV: Submission

The lab requires the following files to be submitted:

- **process.c:** (50 pts)
  You do not need to submit the support files we provided. Your implementation should be entirely contained in *process.c*. Naturally, your code should not depend on any changes in the support files.
- **test_1.c** and **test_2.c**: (20 pts)
  These two files should have a main function and call *process.c* similar to the provide test case. The scoring will depend on the content or the report (testing section) as well.
- **report.pdf**: (10 pts + 10 style)
  Provide a detailed description of your implementation of processes (max 5 pages in 11pt font and single line spacing – we will not read anything after page 5.

  We recommend using illustrations to describe your key data structures (processes, etc.) and key properties of the implementation that you use to ensure correct operation. This write-up should also demonstrate that you understand the files that were provided (*3140.s, 3140_concur.c*) in addition to your own implementation.

- Design
    - A high-level overview of your code. Did you make and design choices on the data-structure? Explain what the provided files do for you, and what they expect in terms of the different process stacks.
- Coding
    - Include any specific details involved in implementing your design. Did you have to make any assumptions, i.e. number of processes, maximum stack use by each process?
- Code Review and Testing.
    - If you followed well-documented techniques (e.g., peer programming, naming conventions, etc.), this is the place to describe it.
    - Describe your testing strategy and how you ensured correct code functionality. For each test case, what did you mean to test in your test and how did you design the test? Where there any things that were difficult to test for?
- Work Distribution
    - How did you collaborate? In your own words, explain how you divided the work, how you communicated with each other, and whether/how everyone on the team had an opportunity to play an active role in all the major tasks. If you used any sort of tools to facilitate collaboration, describe them here. If there were any issues with the team, how did you try to resolve them? Note, you will also get a chance to give individual feedback and rate the collaboration, but we would like you to document what you intended to do here.
- Index
    - Please include any drawings or diagrams to support your work. This section will not count towards the 5-page limit. This layout helps us ensure the page limit and timely grading.

We encourage you to use meaningful variable and function names, comments, etc. to enhance code comprehensibility.

All files should be uploaded to CMS before the deadline. Multiple submissions are allowed, but only the latest submission will be graded.