Lab 4 Locks

Deadlines & Grading

- **Code and report submission**: Friday, March 29 at 9:00 PM on CMS, **groups of 2**
- **Grading**: 8% of total grade

Section I: Overview

Goal. The purpose of this lab is to give you experience with implementing locks, and (optionally) condition variables using the FRDM-K64F microcontroller. To successfully complete this lab, you will need to understand the following,

- Concurrency (Lab 3)
- Locks
- Condition Variables (optional)

NOTE 1 We suggest that you first understand locks and critical sections 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.

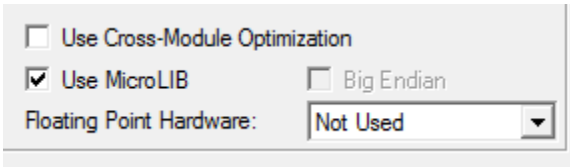
- The micro-controller boards should be handled with care. Misuse such as incorrectly connecting the boards is likely to damage the device.
- These devices are *static sensitive*. This means that you can "zap" them with static electricity (a bigger problem in the winter months). Be very careful about 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 ($35).

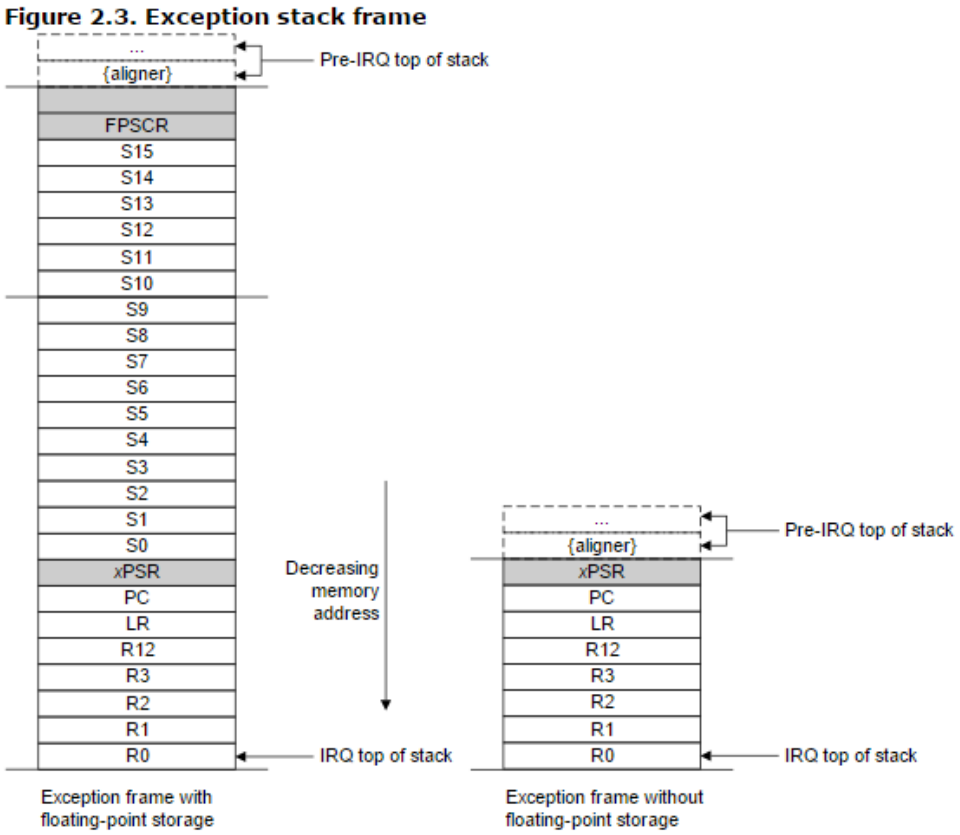
Section II: Getting Started

Part 1: µVision Setup

First set up your project as described in Lab 0 (including setting the scatter file).

The Cortex M4 processor supports two types of stack frames: one with floating-point storage and one without floating-point storage. The image below describes the differences between the two options. To make the code behave uniformly over all lab groups, make sure your implementation uses the exception frame WITHOUT floating-point storage. To do this, navigate to *Target Settings* » *Target* and make sure "Floating Point Hardware" is set to "Not Used", and **that the "Use MicroLIB" option is set**:





Part 2: Adding the Files

The following files provided to you for the lab are the same as in Lab 3:

3140_concur.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.
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*.

The following files are additional for Lab 4 or are modified from Lab 3:

3140.s – contains the assembly language definition of the timer interrupt and functions called when a process terminates.
lock.h – contains the C definitions of functions you must implement for locks
shared_structs.h – contains the C definitions of the structures used across various files
cond.h – contains the C definitions of functions you can optionally implement for condition variables

NOTE 2 *3140.s* has a slight modification from Lab 3 to allow the `process_blocked()` function to be called.

lab4_l0.c – a test program that uses locks
lab4_l1.c – another test program that uses locks
lab4_x0.c – a test program that uses condition variables
lab4_x1.c – another test program that uses condition variables

Add them to your project by right clicking *Source Group 1* in the Project window and clicking on *Add Existing Files to Group 'Source Group 1'...* You may need to change the *Files of type:* in the file selection window to "All files (*.*)" to see all the files.

You should additionally add your *process.c* implementation from Lab 3 to your project. If you are not confident in your implementation of Lab 3, then please contact the TAs to get a sample solution after the lab3 due date.

NOTE 3 Only one of the *lab4_*.c* files can be active in your project at a time, since all of them contain a `main` method. You can either only add one of them to your project at a time and remove it when you want to use another one, OR you can add all of them to your project and right-click on all except one, click *Options for File '<filename>'...*, and uncheck *Include in Target Build*.

Create a new file *lock.c* and (optionally) *cond.c* which will implement the functions needed for the lab. You can create a new file by right clicking *Source Group 1* in the Project window and clicking on *Add New Files to Group 'Source Group 1'...*

NOTE 4 Look through all the provided files to make sure you understand all the code provided. The only provided file that should be modified is *shared_structs.h*. All other functionality can be implemented in files you provide (e.g. *process.c*, *lock.c*, *cond.c*, etc.). Your implementation should only use C and not have to use assembly code other than the routines we have already provided.

Part 3: Code Changes

Shared Structures

Your definitions for the shared data structures should be written in *shared_structs.h*. In Lab 3, we defined `process_state` inside *process.c*. For this lab, you should copy your definition of `process_state` from *process.c* to *shared_structs.h*. In *process.c*, add the following include to the top of the file:

```
#include "shared_structs.h"
```

You may need to add additional fields in `process_state` to allow processes to work with locks. In addition, you should provide the definitions of the `lock_t` and `cond_t` in this file.

Process Blocked

The provided code includes the function `process_blocked()`. When a process attempts to acquire a lock but has to block because some other process holds the lock, you should call `process_blocked()`. Look at the assembly code for its implementation in *3140.s*. The function basically saves the state of the current process and executes a context switch. Note that, in this scenario, `process_select()` will be called with a non-zero argument. You must differentiate between this case and a normal context switch. (**Hint**: save some information into the `current_process` data structure before calling `process_blocked()`).

Your lock implementation code must not have any busy-waiting, i.e., you should never have a process just running in an empty while loop when another process could have been scheduled. `process_blocked()` can be used to ensure this doesn't occur.

In the same spirit, a process that has been blocked on a particular lock should never enter the running state unless some process unlocks the same lock.

Section III: Assignment

Part 1: Implementing Locks

The functions you must implement are defined in *locks.h*. They are:

```
void l_init (lock_t *l);
void l_lock (lock_t *l);
void l_unlock (lock_t *l);
```

A description of what they do is also found in the header file.

You must additionally modify your `process_select` function to detect blocked processes and not run them until the lock is released.

Implement the three functions (`l_init, l_lock`, and `l_unlock`) listed above in *lock.c*. Also, modify `process_select` so that blocked processes don't run.

Part 2: Testing your Implementation

We have posted two test cases for locks. Your lab should pass these tests, and several tests that we will not provide.

You must also provide *two* test cases for your lock implementation to demonstrate that your implementation works correctly. Each test case should focus on some non-trivial aspect of the locking behavior. Name these files *test_l1.c and test_l2.c*.

Section IV: Extra Credit

Part 1: Implementing Condition Variables

The functions you must implement are defined in *cond.h*. They are:

```
void c_init (lock_t *l, cond_t *c);
void c_wait (lock_t *l, cond_t *c);
void c_signal (lock_t *l, cond_t *c);
int c_waiting (lock_t *l, cond_t *c);
```

A description of what they do is also found in the header file. Your implementation must not have any busy-waiting. To qualify for extra credit, your submission must also contain *cond.c* with your implementation of condition variables, and your documentation must also describe the implementation of condition variables (you can use up to two more pages if necessary).

NOTE 5 We strongly recommend to first make your implementation of locks (without condition variables) work correctly before working on your condition variable implementation; in other words, make sure that simply ignoring cond.c would be a valid Lab 4 submission.

Implement the four functions (`c_init, c_wait, c_signal, c_waiting`) listed above in *cond.c*.

Part 2: Testing your Implementation

Your lab should pass all the posted tests, and several tests that we will not provide.

You must also provide *two* tests cases for your conditional variable implementation to demonstrate that your implementation works correctly. Each test case should focus on some non-trivial aspect of the condition variable behavior. Name these files *test_c1.c and test_c2.c*.

Section V: Submission

The lab requires the following files to be submitted:

- **concurrency.zip**
  The entire implementation and test cases, including the files we have provided. Do not include µVision files.

- **report.pdf**:
  Provide a detailed description of your implementation of processes and locks (max 5 pages in 11pt font and single line spacing. However, refrain from adjusting the size of periods or margins).

  We recommend using illustrations to describe your key data structures (processes, locks, etc.) and key properties of the implementation that you use to ensure correct operation.

  You should also include a section on how you tested your implementation and describe the test cases you provide.

  We require that you describe in detail, in a separate section titled "Work Distribution", the following:
  a) How did you carry out the project? In your own words, identify the major components that constituted the project and how you conducted the a) design, b) coding, c) code review, d) testing, and e) documenting/writing. If you followed well-documented techniques (e.g., peer programming), this is the place to describe it.
  b) 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.

We encourage you to use meaningful variable and function names, comments, etc. to enhance code comprehensibility. All code files should be commented in a way that makes it clear why your program works.

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