Introduction to POSIX

The goal of this lab session is to discover some constructs of the POSIX API through simple exercises.

Using code for this lab session

All source code is provided in the archive `posix_lab.tar.gz`. To open this archive, issue the following command "`tar zxvf posix_lab.tar.gz`" this will create the directory `posix_lab` with all source code.
A makefile is provided to compile all source code. Issue "`make help`" for more details.
When compiling, yoy may see in the console: `foo.c:9: note: #pragma message: TODO - Exercise X`.
This indicates a place where you need to correct the source code for a given exercise.
Note: the code provided for this lab session has been tested for Linux OS only. It relies on mechanisms from Unix and POSIX libraries that are not available on Mac OS X or Windows.

Exercises around the POSIX API

The goal of this first lab session is to manipulate the POSIX concurrent API: threads, mutexes, condition variables to implement basic concurrency patterns.

Exercise 1 (Simple thread manipulation) *In this exercise, we want to experiment with thread creation/finalization. Source code from `ex1.c` is an empty skeleton to be completed.*
Review this source code, and complete it to

- *create threads;*
- *proper finalize each thread;*
- *block the environement thread until all threads are finalized using `pthread_join()`.*

Exercise 2 (About thread termination) *In the previous exercise, using `pthread_join()` was unsatisfactory: this function takes as parameter the id of the thread to wait for.*
In this exercise, we want to get rid of this constraint: the environment thread will be notified by the thread itself when it finishes. We will implement this capability using condition variables.
Source code from `ex2.c` is a skeleton to be completed.
Review this source code, and complete it to

- *create threads;*
- *when a thread finalizes, notify the environment thread using `pthread_cond_signal()`;*
- *block the environement thread until all threads are finalized.*

Your implementation report shall document how the finalization scheme you propose is correct. Be sure you test various scenarios.

Exercise 3 (Read/Write Lock) *In this exercise, we want to implement a read/write lock. Such lock allows multiple readers to read a data, but allows at most one writer to update it.*
The challenge when designing an implementation of a read/write lock is to ensure there is no starvation.
Source code from `ex3.c` is a skeleton to be completed.
Review this source code, and complete it to implement a read/write lock. We will use a pattern combining a mutex and two condition variables. Condition variables will be used to control the queue of readers and writers. The detail of the strategy is presented in the source code as comments.
Your implementation report shall document how the implementation scheme you propose is correct. Be sure you test various scenarios, by modifying the number of readers and writers.

Exercise 4 (Railroad crossing) *In this exercise, we consider a part of railroad in which there is a single track over which only one train at a time is allowed.*
A semaphore is guarding access to this track. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter." We consider that two tracks are connected to the single track on both side, see the picture below.

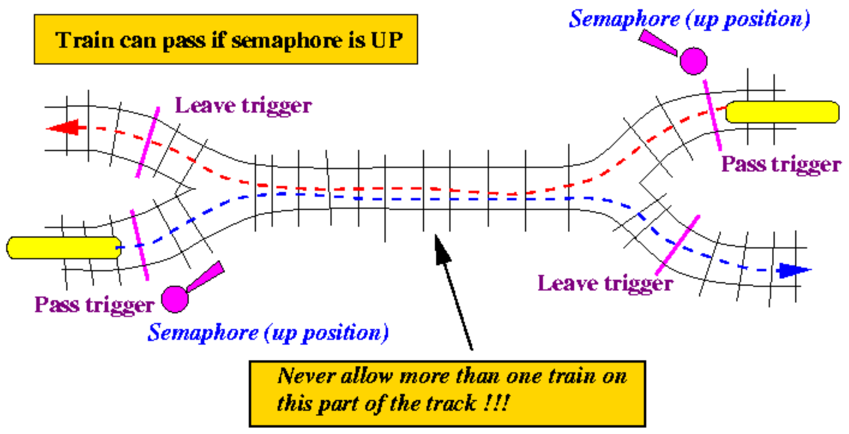

Figure 1: The

We propose a C program to support this behavior, see `train.c`. You are tasked to complete element of this program. Note that this program is more complex that the picture, as we need to carefully synchronized many items.

- *What is the role of the semaphore G ? How to initialize it ?*
- *What is the role of the array of semaphores F ? How to initialize it ?*
- *We define an array of threads trains to represent each train. Write the source code to initialize this array. Each thread has to execute the `train_job()` function.*
- *Complete this program so that the environment thread waits for the completion of all threads.*
- *What is the purpose of the test marked at comment #1 ? Complete the code associated to comment #2.*
- *Is the program safe ? what misbehaviors could happen ? Propose a patch to the program that addresses it.*
- *Is the solution proposed fair to all trains ? Detail your answers with a complete scenario.*