CS2106: Operating Systems
Lab 3 – Solving Synchronization Problems in UNIX

Important:

- The deadline of submission through LumiNUS is Wednesday,**14th October 2pm**
- The total weightage is 7**% + [Bonus 1%]:**
    - Exercise 1: 1% [1% demo]
    - Exercise 2: 2%
    - Exercise 3: 2%
    - Exercise 4: 2%
    - Exercise 5: 1% **bonus**
- You must ensure the exercises work properly on the SoC Compute Cluster, hostnames: xcne0 – xcne7, Ubuntu 20.04, x86_64, GCC 9.3.0.

Section 1. Exercises in Lab 3

There are **five exercises** in this lab. The purpose of this lab is to learn about thread synchronization in Linux by solving some synchronization problems and implementing your own constructs. Due to the difficulty of debugging concurrent programs, the complexity of this lab has been adjusted accordingly. Hence, you should find the required amount of coding "minimal".
General outline of the exercises:

- Exercise 1: Implementing a simple barrier
- Exercise 2: Multithreaded implementation for the FizzBuzz problem
- Exercise 3: Synchronizing trains entering loading/unloading bays
- Exercise 4: Synchronizing trains exiting loading/unloading bays
- Exercise 5: Generalizing the train problem to multiple priorities

Section 2. Barrier Synchronization

You are required to implement a synchronization construct called a barrier. A barrier for a group of threads (or processes) in the program means any thread must stop at the barrier and cannot proceed until all other threads/processes reach the barrier.

You can use `POSIX semaphores` to implement your barrier(s). You are not allowed to use any other synchronization mechanism provided by `pthread` library (or other libraries) for exercises 1 and 2.

Exercise 1: Simple Barrier [demo – 1%]

You are required to write a simple barrier to synchronize n threads. All n threads stop (block) at the barrier and wait for all other threads to reach the barrier before they can continue their execution. This barrier is used only one time by all threads.

The skeleton code provided is designed as follows:

- `ex1_runner` takes command line arguments to set the number of threads that the program starts
- `ex1_runner` starts n threads
- each thread prints some information and waits at the barrier
- threads continue their execution and exit
- threads are joined and the program execution ends

You are required to implement this single-use barrier called by each thread one time to ensure that all threads synchronize and wait for each other to reach the barrier.

The implementation should be provided in file `barrier.h` and `barrier.c`. You are required to add any elements to the `barrier_t` structure and to implement the following functions:

- `void barrier_init (barrier_t *barrier, int count)` – initialize the barrier to be used by `count` threads. This function is called in `ex1_runner.c` before the threads are started.
- `void barrier_wait (barrier_t *barrier)` – waits at the barrier until all `count` threads reach the barrier. This function is called in `ex1_runner.c` by each thread.
- `void barrier_destroy (barrier_t *barrier)` – cleans up any structures used by the barrier. This function is called in `ex1_runner.c` after the threads are joined.

For this exercise, you can modify:

- **`barrier.h`**
- **`barrier.c`**

`ex1_runner.c` should not be modified as it might be replaced during grading with another file.

To compile the files for exercise 1 you can use the `Makefile` provided in folder `ex1-2`:

`> make clean`	→ cleans up the folder by removing executables and object files.
`> make all`	→ compiles ex1_runner and ex2_runner
`> make ex1_runner`	→ compiles ex1_runner
`> make ex2_runner`	→ compiles ex2_runner

Exercise 2: Multithreaded FizzBuzz [2%]

You are required to write a synchronized, multithreaded implementation of the FizzBuzz problem.

The FizzBuzz problem is described below:

> Write a program that prints the numbers from 1 to n (increasing order, including 1 and n). But for multiples of three print Fizz instead of the number and for the multiples of five print Buzz. For numbers which are multiples of both three and five print FizzBuzz.

The skeleton provides a runner program, `ex2_runner.c`, which spawns 4 threads, and calls `num_thread`, `fizz_thread`, `buzz_thread`, `fizzbuzz_thread` in each of the threads, respectively. These four functions are called with a printing function as argument. You **must** use the function received as argument to perform the printing.

In `ex2_runner.c`, each of the four thread functions receives one of the following printing functions, respectively: `print_num`, `print_fizz`, `print_buzz`, `print_fizzbuzz`. Calling a specific printing function by name or simply printing from the thread instead of using the function pointer provided as argument to the thread function will bring deductions, as we will replace these functions during grading. Both the name and the implementation of the printing functions might change.

Your task is to implement the multithreaded solution of the FizzBuzz problem by modifying `fizzbuzz_workers.c`:

- `void fizzbuzz_init (int n)` – initialises the resources needed for your implementation. This function is called from ex2_runner.c before the threads are spawned.
- `void num_thread (int n, void (*print_num)(int))`
- `void fizz_thread (int n, void (*print_fizz)(void))`
- `void buzz_thread (int n, void (*print_buzz)(void))`
- `void fizzbuzz_thread (int n, void (*print_fizzbuzz)(void))`
  Each thread function uses the `function` received as function pointer `print_*` to print information for the numbers from `1` to `n`.
- `void fizzbuzz_destroy ()` – cleans up resources initialised for the implementation. This function is called from ex2_runner.c after the threads are joined.
- You can define global variables in file `fizzbuzz_workers.c` to be shared among threads.

Check `ex2_runner.c` to understand better how the threads are created and how the *`_thread` functions are used. The expected result would be to see the `numbers`, Fizz, Buzz and FizzBuzz printed in increasing order (from 1 to n, inclusive).

For this exercise, you can modify:

- **`fizzbuzz_workers.c`**

`ex2_runner.c` and **`fizzbuzz_workers.h`** should not be modified as they might be replaced during grading with other files.

Section 3. Synchronization of Trains

You are required to implement the synchronization of trains entering and exiting from multiple loading/unloading bays. Trains arrive at the loading bays on one railway (one-way) and leave the bays using another railway (one-way). If all the bays are used by other trains, the incoming queue on the entry line. Figure 1 shows the diagram of a loading station with 5 loading bays.

You need to synchronize the trains to enter the bays or queue for loading/unloading on the entry line (railway), and exit the bay when there is no other train on the exit line (railway). Loading/unloading a train takes a random time, and the trains have different classes of priorities.

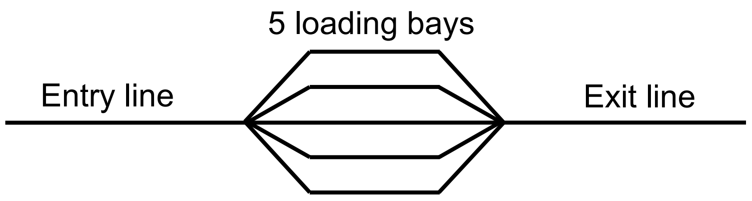

Figure 1: Entry and Exit from 5 Loading Bays

In the skeleton provided, we implemented a train runner program (`train_runner.c`), Makefile and additional files needed for exercises 3 and 4. Each train is simulated using a thread that executes function `simulate_train`, and you need to use synchronization mechanisms to implement the train synchronization for entering and exiting the bays. The trains receive at creation time a random priority class and a random loading/unloading time. A train will:

- enter the loading bay from the entry line
- sleep for some time in the bay waiting for loading/unloading
- exit the loading bay to the exit line

The program `train_runner` takes as input the command line arguments:

- `no_of_trains` – showing how many trains (threads) the program will start.
- `no_of_bays` – showing the number of bays available for loading/unloading - maximum 50 bays.
- `no_of_priorities` – showing the number of priorities. Each train priority will be between 0 (inclusive) and no_of_priorities (exclusive) - maximum 50 priorities.
- seed – an optional argument used to generate the loading time and the priority class.

The creation and joining of threads are implemented for you in `train_runner.c`. You are required to implement the synchronization functions called by a train to enter a bay and exit the bays. Implement your code in `entry_controller.c and exit_controller.c`:

- entry controller, implemented in `entry_controller.c` is called by function `enter_loading_bay` and `exit_loading_bay` in `train_runner.c`
- exit controller, implemented in `exit_controller.c` is called by function `exit_loading_bay` in `train_runner.c`

Exercise 3: Entering the Bay [2%]

The trains arrive on the entry line one at a time by calling function `enter_loading_bay`. The **entry rules** follow:

- If there is any free bay when a train arrives on the entry line, the train enters that bay.
- If all the bays are used by other trains to load/unload, the incoming train is placed in a queue on the entry line.
- Once there is a free bay, the first train on the entry line can proceed.
- The trains queueing on the entry line enter the bays in a first-in-first-out order.

You do not need to keep track of which bay a specific train is loading/unloading on. Entering the bay is achieved by calling `wait` on an `entry_controller`.

You are required to synchronize the trains entering the bay according to the **entry rules**. You can modify the files `entry_controller.h` and `entry_controller.c`. Use the structure `entry_controller_t` to declare any synchronization constructs and data structures needed to synchronize trains at the entrance to the loading bays. Next, implement the following functions:

- `void entry_controller_init (entry_controller_t *entry_controller, int no_of_bays)` – initializes the entry controller with `no_of_bays` bays. This function is called exactly once in `train_runner` main function.
- `void entry_controller_wait (entry_controller_t *entry_controller)` – waits until there is a free bay and the train can enter the bay. Trains (threads) enter the bay (successfully complete the wait) according to the **entry rules**. This function is called from the `enter_loading_bay` function for each thread.
- `void entry_controller_post (entry_controller_t *entry_controller)` – signals that the train (thread) left the loading bay. This function is called from `exit_loading_bay` function.
- `void entry_controller_destroy (entry_controller_t *entry_controller)` – cleans up any structures you might have used for your controller. This function is called exactly once after all trains are joined in `train_runner.c`

The implementation of the `entry_controller_wait` should not be done using busy waiting. The maximum number of trains that will try to enter the bay at the same time (queueing on the entry line) is 5000.

You might use any synchronization mechanism provided by `pthread` library, in addition to POSIX semaphores. However, there are solutions to this problem that only use semaphores.

For this exercise, you can modify:

- **`entry_controller.h`**
- **`entry_controller.c`**

`train_runner.c` should not be modified as it will be replaced during grading with another file.

Exercise 4: Exiting the Bay [2%]

Once the trains are done loading/unloading, they exit the bay by calling function `exit_loading_bay`. Since there is only one exit line, at most one train at a time can leave the bay and start moving on the exit line (critical section). The trains move for a random amount of time on the exit line until the exit line becomes free again and can be used by another train. The trains have different priorities, and they need to give priority to other higher priority trains when exiting the bays, according to the following **priority rules:**

- If two trains of the same priority are waiting to exit the bay, once the exit line is free, any of them can proceed.
- If two trains of different priorities are waiting to exit the bay, once the exit line is free, the train with the highest priority should proceed.

For this exercise, the trains can have at most two priority classes:

- priority 0 is the higher priority
- priority 1 is the lower priority.

You are required to synchronize the trains exiting the bay according to the **priority rules**. You can modify the files `exit_controller.h` and `exit_controller.c`. Use the structure `exit_controller_t` to declare any synchronization constructs and data structures needed to synchronize trains at the exit from the loading bays. Next, implement the following functions:

- `void exit_controller_init(exit_controller_t *exit_controller, int no_of_priorities)` – initializes the controller for trains with `no_of_priorities` priorities to exit the bay. This function is called exactly once in `train_runner` main function with a value of 2 for `no_of_priorities`.
- `void exit_controller_wait(exit_controller_t *exit_controller, int priority)` – blocks and waits to gain access to the exit line according to the priority rules. This function is called in `exit_loading_bay` function.
- `void exit_controller_post(exit_controller_t *exit_controller , int priority)` – signals that the exit line is free, and the train that left the exit line had priority `priority`. This function is called in `exit_loading_bay` function.
- `void exit_controller_destroy(exit_controller_t *exit_controller)` – cleans up any structures you might have used for your controller. This function is called exactly once after all trains are joined in `train_runner.c`.

The implementation of the `entry_controller_wait` should not be done using busy waiting.

You might use any synchronization mechanism provided by pthread library, in addition to POSIX semaphores. However, it is possible to write a solution to this problem that uses only semaphores.

For this exercise, you can modify:

- **`exit_controller.h`**
- **`exit_controller.c`**

`train_runner.c` should not be modified as it might be replaced during grading with another file.

To compile the files for exercises 3 and 4 you can use the `Makefile` provided in folder `ex3-4`:

`> make clean`	→ cleans up the folder by removing executables and object files.
`> make all`	→ compiles train_runner

Please note that `train_runner.c` does not check the correctness of your implementation for your `entry_controller` and `exit_controller`. Successfully running the `train_runner` does not guarantee full marks for exercises 3 and 4.

Exercise 5: General Priority Controller [Bonus 1%]

In exercise 4, you implemented a priority controller for trains with 2 priority classes. In exercise 5, you are expected to implement a priority controller for any number of priority classes. The new controller should follow the priority rules mentioned under exercise 4.

Test your controller using `exit_loading_bay` function in our train synchronization problem. Place your solution files in folder `ex5`. Write your implementation for the priority controller under file `exit_controller.c`.

For this exercise, you can modify:

- **`exit_controller.h`**
- **`exit_controller.c`**

Place your files in a new folder called **`ex5.`**

Section 4. Submission

Zip the following folders as E0123456.zip (**use your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix**):

```
b. ex1-2/
     -  barrier.h
     -  barrier.c
     -  fizzbuzz_workers.c
c. ex3-4/
     -  entry_controller.h
     -  entry_controller.c
     -  exit_controller.h
     -  exit_controller.c
d. ex5/ [only if attempted]
     -  entry_controller.h
     -  entry_controller.c
     -  exit_controller.h
     -  exit_controller.c
```

Do **not** add additional folder structure during zipping, e.g. do not place the above in a "lab3/" subfolder etc.

Upload the zip file to the "Lab Assignment 3" folder on LumiNUS. Note the deadline for the submission is **14th October, 2pm**.

Please ensure you follow the instructions carefully (output format, how to zip the files etc). **Deviations will be penalized.**