CS2106: Operating Systems
Lab 3 – Thread Synchronization in UNIX

Important:
- The deadline for LumiNUS submission: **March 19th, 2359**
- The total weightage: **7%**
  - Exercise 1: 1% **[Lab Demo Exercise]**
  - Exercise 2: 1.5%
  - Exercise 3: 1.5%
  - Exercise 4: 3%

Reminder:
- Ensure your code works properly on the SoC Compute Cluster Nodes.
- Ensure your submission follow the specified format.

Please start on this lab early. This lab is quite challenging as it is dealing with the "hardest" topic in CS2106, i.e. synchronization. Although you will not write a lot of code, you need to understand multi-threading, mutex and semaphore (and their corresponding library calls). You also need ample time to understand and tackle the synchronization problem. Consider this a "friendly warning" ☺

Section 0. Basic Information

Please refer to the "Introduction to OS Labs" document if you have forgotten how to unpack / setup the lab exercises.

You only need to do this if you have just formed a team for lab 03. If you have already registered a team for previous labs, there is **no need** to re-register. Please use the google form to register your team:

https://forms.gle/2XGm3zFYpXdFTgpc6

Select your name and your buddy's name from the dropdown list to form a team. Your team will be in effect **from this lab onward**.

Section 1. Exercises in Lab 3

There are **four exercises** in this lab. The purpose of this lab is to learn about thread synchronization in a Unix-based OS. 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: Barrier
- Exercise 2: Basic readers-writers problem
- Exercise 2: A more fair readers-writers problem
- Exercise 3: Synchronization in a roundabout

Section 2. Exercise 1: Simple Barrier [Lab Demo Exercise]

As discussed in tutorial 5 question 4, **barrier** is a useful synchronization mechanism for multiple tasks. We are going to implement a simple barrier to synchronize `n` threads for this exercise. 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 given skeleton code `ex1.c` works as follows:

1. It takes in command line arguments `n` as the number of threads.
2. The program creates `n` threads:
    i. Each thread prints some information and waits at the barrier.
    ii. Each thread continue their execution and exit.
3. All `n` threads are joined (i.e. "waited") 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.

> Use **only** Posix General Semaphores to implement the barrier. Include the library `<semaphore.h>` and take a look at the relevant library functions: `sem_init()`, `sem_wait()` and `sem_post()`.

The implementation should be provided in file `barrier.h` and `barrier.c`. You are required to:

- Add necessary field variables to the `barrier_t` structure.
- 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.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.c` by each thread.
`void barrier_destroy (barrier_t *barrier)` Cleans up any structures used by the barrier. This function is called in `ex1.c` after the threads are joined.

For this exercise, you should modify only **`barrier.h`** and **`barrier.c`**.

To compile exercise 1 you can use the `Makefile` provided in folder **`ex1/`**:
`$ make` → compiles **`ex1`** with the barrier code together

If you need to clean up the object files and executable, you can use:
`$ make clean` → cleans up the folder by removing executables and object files.

Section 3. Readers-Writers Lock

Exercise 2 and 3 looks at the readers-writers synchronization problem. Exercise 3 is an enhanced version of exercise 2. For these exercises, it is sufficient to use **`pthread_mutex`** imported from the **`pthread`** library as the synchronization mechanism.

3.1 Readers-Writers Basic Version

In this problem, there are two kinds of threads – **reader threads** and **writer threads**, which are trying to read from or write to a shared resource respectively. We are going to implement the **readers-writers lock** (**RW-Lock**) needed to fulfil the basic requirements of the **readers-writers problem**, as given below:

1. A **writer** cannot write to the shared resource when anyone else (reader or writer) is using the resource.
2. **Readers** cannot read from the shared resource when a writer is writing to it. However, multiple readers can read the shared resource simultaneously.

You are given the following files for this exercise. More details are given in subsequent pages.

`ex2_driver.c`	A "driver" program that creates and run the reader and writer thread. [No need to modify]
`rw_lock.h`	Header file containing the **`rw_lock`** structure declarations and related functions prototypes. [You need to modify the **structure declaration** only, i.e. do not change the function prototypes.]
`ex2.c`	Implementation files for the **`rw_lock`** related functions. [You need to complete the function as needed. Do NOT change the function header (i.e. name, return type and parameters of the functions should not change.)]

In **`ex2_driver.c`**, you will find the following global variables:

- **`value`** – The "shared resource" to be written and read from
- **`reader_count`** – The number of readers accessing the shared resource. You should be using this to track the number of readers.
- **`writer_count`** – The number of writers accessing the shared resource. You should be using this to track the number of writers.
- **`max_concurrent_readers`** – The maximum number of concurrent readers reached in your program.
- **`max_mutex`** – The mutex to update the **`max_concurrent_readers`**.

You will also find the following functions:

`main()`

This is the main driving function for the program. In this function, the following would happen:

1. **`initialise(read_write_lock)`**, which creates and initializes your lock, so it is ready for use as a RW-lock.
2. **`WRITERS`** writer threads are created and initialized, each running the **`writer`** function.
3. **`READERS`** writer threads are created and initialized, each running the **`reader`** function.
4. The created threads are all joined and the result of the program is printed.
5. **`cleanup(read_write_lock)`** is then run to perform required resource cleanup.

`writer()`

This is the main function call for each writer thread created. It runs **`WRITE_COUNT`** loops, each loop performing the following:

1. **`writer_acquire(read_write_lock)`**
2. Checks if the conditions are valid for writing. If not, an error will be registered for this thread.
3. Writes its **`threadid`** to the shared resource.
4. **`writer_release(read_write_lock)`**

`reader()`

This is the main function call for each reader thread created. It runs **`READ_COUNT`** loops, each loop performing the following:

1. **`reader_acquire(read_write_lock)`**
2. Checks if the conditions are valid for reading. If not, an error will be registered for this thread.
3. Checks for number of other readers accessing the shared resource and updates the maximum.
4. Reads the shared resource.
5. **`reader_release(read_write_lock)`**

Your task:

Currently, the structure declaration of **rw_lock** (in **rw_lock.h**) and its related functions (declared in **rw_lock.h** and implemented in **ex2.c**) are incomplete.

Your task is to amend **ex2.c** and **the rw_lock structure** in **rw_lock.h** to solve the readers-writers problem. **rw_lock** should fulfil the following requirements:

- **Correctness:** The program is ensured to run correctly, according to the rules of the readers-writers problem as mentioned above.
- **Concurrency:** The **Max Concurrent Readers** is maximized. (What is the highest possible number?)

Your program should be able to run correctly with at least **5 writers** and **5 readers**, with **50** writes and **50** reads per writer and reader respectively.

> Reminder:
> You only need to change the **rw_lock structure declaration** in **rw_lock.h** and complete the function implementations in **ex2.c.** Only changes you made in these two files will be used for grading. You may change the other files provided (**ex2_driver.c**) during your own testing, but note that they will be replaced with the original files when we test your assignments.

To compile your programs in **ex2/** folder:

```
$ gcc -Wall ex2_driver.c ex2.c -o ex2 -lpthread
            //builds ex2 executable using 2 c files
            //-Wall  shows all warnings
OR

$ make      //builds ex2 according to Makefile
```

An example run:

```
$ ./ex2 5 5 50 50
            //run ex1 with 5 readers and 5 writers with 50 operations each
```

If the program terminates **correctly**, the following output is expected:

```
SUCCESS!
Total writes: 250, Total reads: 250, Max Concurrent
Readers: 5
```

Otherwise, you would see this if **correctness is not fulfilled**:

```
Program failed: 4 bad threads found.
```

3.2. Exercise 3 – Fairer Readers-Writers

Consider the following sequence of events which illustrate the drawback of the basic version implemented in exercise 2:

```
1. Reader 1 requests access.
2. Reader 2 requests access.
3. Writer 1 requests access. [Blocked by Readers]
4. Reader 1 leaves.
5. Reader 1 requests access.
6. Reader 2 leaves.
7. Reader 2 requests access.
   …
```

Even if we ensure the basic conditions are met, notice that writers can be **starved** for a very long time when such sequences of actions occur.

You are given similar set of files as in exercise 2:

- Exercise 3 driver program **`ex3_driver.c`** is almost identical to **`ex2_driver.c`**. The only difference is that the **`reader`** threads are created before the **`writer`** threads.
- You have **`rw_lock.h`** and **`ex3.c`** with similar purpose as in exercise 2.

You can compile and run **`ex3.c`** similarly to **`ex2.c`**. i.e. using the full compilation command or the "**`make`**" command.

Your task:

For this exercise, you can start from your solution in exercise 2. A correct exercise 2 solution would unfortunately result in the following output (why?):

```
Program failed: All writing operations happen after
reading.
```

This is because in the new driver program, readers are created first and can "hog" the shared resource before any writers can do anything.

Your task is to amend the same files (**`ex3.c`**, **`rw_lock.h`**), so that:

- All requirements from Exercise 2 are still fulfilled. (**Correctness** and **concurrency**)
- **Less Writer Starvation:** Each writer gets to write before all the readers finish. (Is there a way to guarantee no starvation? Under what circumstances can a writer still be starved?)

Reminder:
You only need to change the **`rw_lock` structure declaration** in **`rw_lock.h`** and complete the function implementations in **`ex3.c`.** Only changes you made in these two files will be used for grading. You may change the other files provided (**`ex3_driver.c`**) during your own testing, but note that they will be replaced with the original files when we test your assignments.

If the program terminates **correctly and without writer starvation**, the following output is expected:

```
SUCCESS!
Total writes: 50, Total reads: 250, Max Concurrent
Readers: 5
```

Otherwise, you would see this if **correctness is not fulfilled**:

```
Program failed: 4 bad threads found.
```

Or, you would see this if **less writer starvation is not fulfilled**:

```
Program failed: All writing operations happen after
reading.
```

Additional resources for synchronization problems

For a detailed and extended view on many synchronization problems check the following book: http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf
Specifically, for readers-writers problem you can check section 4.2.

Section 4. Roundabout Synchronization

For exercises 4, we will use **general semaphores**, which can be included with library **`<semaphore.h>`.**

In this exercise, we are going to design a traffic synchronizer to prevent crashes in a single-lane roundabout.

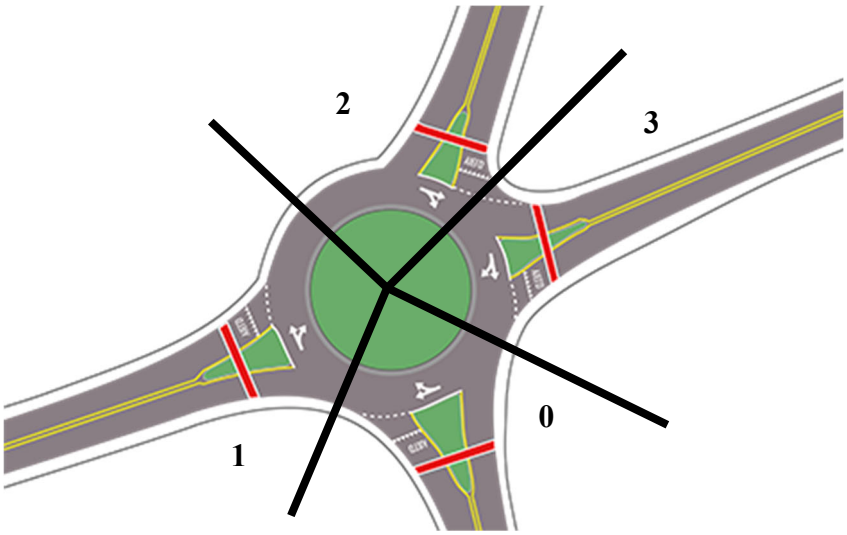

Figure 1: Roundabout with segments and streets

Figure 1 shows a roundabout with 4 streets connected to it. The roundabout is divided into multiple segments, each segment is connected to a street. In a roundabout with **`n`** streets, the segments and streets are numbered **`0`** to **`n - 1`** in clockwise fashion.

Cars enter the roundabout from a street and leave at another street. Once it enters the roundabout, the car occupies the corresponding segment. The car advances through the roundabout by moving to the next segment (clockwise). To avoid crashes in the roundabout, there can be **at most one car in each segment at any given time**.

When a car is in a roundabout, it can only move **clockwise** to the road segment adjacent to it. In the example above, if a car enters at **`3`** and it is supposed to leave the roundabout at segment (street) **`1`**, it will advance through segment **`0`** and then **`1`**, followed by leaving the roundabout. The car will always leave the roundabout once it reaches its exit (a car should not circle through the roundabout). U-turn should not be possible (the entry is different from the exit).

You must enforce the "**one car per segment rule**":

1. If there is a car at segment **`n`**, no other car can enter the roundabout from street **`n`** or move into that segment (from segment **`n-1`**) until that car moves or exits.
2. Multiple cars may wait to enter the roundabout from street **`n`**, but only one of them can enter when there is no car in segment **`n`**.

3. A car may exit at segment **n** even when there are cars waiting to enter at that street (two-way street). A car leaves the segment before another car can enter at the same street (one lane roundabout).

In addition, we are going to use "**equal priority**" rule. Cars waiting to enter at a segment **n** (from street **n**, or from the previous segment in the roundabout) have equal chances of moving in segment **n** once there is no car in that segment. This means that cars that are in the roundabout have the **same priority** with cars waiting to enter from the street.

As this is a roundabout in Singapore, there are many impatient drivers – your traffic synchronizer must not only ensure no crashes, but also maximize the number of cars moving at once. Don't keep the drivers waiting when they can move!

Files given for this exercise

The file **ex4_driver.c** is the main driving program for this exercise. In this program,

1. The information about each car is stored using a structure. These cars are initialized with their entry and exit segments.
2. Other variables are initialized. There are various counters to keep track of
   - The number of cars in a segment (used to check the one car per segment rule)
   - The maximum number of cars moving simultaneously
   - The maximum number of cars in the roundabout (at the same time)

   [Semaphores are used to avoid synchronization problems when updating these counters.]
3. A car entering and moving through the roundabout is represented using a thread. **num_of_cars** threads are created. Each car thread executes function `car` (to be implemented by you in file **ex4.c**).
4. The threads are joined (i.e. waited for their completion).
5. Final message is printed.
6. Clean up an exit.

ex4_driver.c implements 3 functions that must be called from the car thread:

1. **void enter_roundabout(car_struct* car) -** move the car from the street to the entry segment. The current segment of the car is updated to the entry segment after successful execution.

2. **void move_to_next_segment(car_struct* car) -** moves a car from current segment to the next segment. The current segment of the car is updated after successful execution.

3. **void exit_roundabout(car_struct* car) -** move the car from the current segment to the exit street. The current segment of the car must be the same with the exit segment for successful execution.

Your task:

Complete the following functions in **`ex4.c`**:

`initialize()` **`cleanup()`**	(if needed) Initialization for each car thread. Clean up code for each car thread before the thread terminates.
`car()`	Your main bulk of work for this exercise.

As mentioned, each car is modelled using a thread. Each car thread executes the function **`car()`**, which has the following basic steps:

1. Enter the roundabout by calling **`enter_roundabout()`** function
2. Move through the roundabout from one segment to another using **`move_to_next_segment()`** function.
3. Exit the roundabout by calling **`exit_roundabout()`** function.
4. Finish the thread execution.

Use synchronization mechanisms (semaphores) to enforce the "**one car per segment rule"** with "**equal priority"**.

Your implementation should meet the following requirements:

- **Correctness:** The program is ensured to run correctly, without crashes or deadlocks in the roundabout.
- **Concurrency:** The **number of concurrently moving cars** in the roundabout is *maximized*. (What is the highest possible number?)
- **Scalability:** Your program should be able to run correctly with at least **20** segments, with **100** cars entering per segment.

[Note that your implementation **must meet the first two requirements before scalability is considered.**]

To compile your programs in **`ex4/`** folder:

```
$ gcc -Wall -DDEBUG ex4_driver.c ex4.c -o ex4 -lpthread
            //builds ex4 executable using 2 c files
            //-Wall  shows all warnings; -DDEBUG turn on the debug flag
Or
$ make
```

An example run:

```
$ ./ex4 10000 5 20
      //run ex4 using <10000> as random seed, <5> segments roundabout and
      //  <20> cars to enter per street.
```

Reminder:
You only need to change **ex4.c**. Only changes you made in **ex4.c** and will be used for grading. You may change the other files provided (**ex4_driver.c**) during your own testing, but note that they will be replaced with the original files when we test your assignments.

Hint for Exercises 4: Complex synchronization problems can often be decomposed into simpler, well-known ones.

Section 5. Submission

Zip the following folders and files as `A0123456X.zip` (**use your student id with "A" prefix and the alphabet suffix**). Remember to modify the archive name and the content accordingly as needed.

a. **ex2/**
   - **ex2.c**
   - **rw_lock.h**

b. **ex3/**
   - **ex3.c**
   - **rw_lock.h**

c. **ex4/**
   - **ex4.c**

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

3.1 Do a quick self-check

We have provided a self-checking shell script to help validating your zip archive. The script checks the following for this lab:

a. The name or the archive you provide matches the naming convention mentioned above.
b. Your zip file can be unarchived, and the folder structure follows the structure described above.
c. All files for each exercise with the required names are present.
d. Each exercise can be compiled and/or executed.

Note that we won't be able to run test cases due to the nature of this lab. Once you have produce the zip file, you will be able to check it by performing the following steps at the **L3/** folder:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh A0123456X.zip (replace with your zip file name)
```

The **`check_zip`** script will print out the status of each of the checks. Successfully passing checks enable the lab TA to focus on grading your assignment instead of performing lots of clerical tasks (checking filename, folder structure, etc). Please note that **points might be deducted if you fail these checks.**

Expected Successful Output

```
Checking zip file....
Unzipping file: A0123456X.zip
Transferring necessary skeleton files
ex2: Success
ex3: Success
ex4: Success
```

3.2 Upload to LumiNUS before deadline!

After verifying your zip archive, please upload the zip file to the "Student Submission→Lab 3" file folder on LumiNUS. Please note that **late penalty is quite steep,** so submit early to avoid last minute issues.

Also, remember that if you are in a team, **only one member need to submit into LumiNUS.** The archive name should be in the format `A0123456X.zip`, where `A0123456X` stands for the student number of *any of the team members*; we will do "background processing" to match with your team members automatically. Both team members will receive the same score for the lab.

~~~ End of Lab 3 ~~~
~~~ End of Lab 3 ~~~
~~~ End of Lab 3 ~~~ ;-)