CS2106: Operating Systems
Lab 3 – Thread Synchronization in UNIX

Important:

- The deadline of submission through LumiNUS is **16th October 2pm**
- The total weightage is **6% + [Bonus 1%]:**
  - Exercise 1: 2%
  - Exercise 2: 1% **[Lab Demo Exercise]**
  - Exercise 3: 3% + **[Bonus 1%]**

Section 1. Exercises in Lab 3

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

Section 2. Readers-Writers Lock

For exercises 1 & 2, it is sufficient to use **`pthread_mutex`** imported with the **`pthread`** library.

2.1 Exercise 1

In exercise 1, we will implement the readers-writers lock (RW-Lock) needed to fulfil the basic requirements of the readers-writers problem. Recall that in this problem, there are two kinds of threads – reader threads and writer threads. Each thread is trying to read from or write to a shared resource respectively. The requirements of this problem are:

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.

Exercise 1 Template

ex1_runner.c is the main driving file for this exercise. In it, 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)`**

ex1.c contains implementations of the functions for **`rw_lock`**. You have to change the code found in this file.

rw_lock.h and **rw_lock_struct.h** are header files defining the function declarations and struct declarations for **`rw_lock`**, respectively. These header files allow the compiler to know the functions are to be "imported" and used by **ex1.c** to **ex1_runner.c**.

Your task for Exercise 1:

Currently, the implementations of **`rw_lock`** (in **rw_lock_struct.h**) and its related functions (declared in **rw_lock.h** and implemented in **ex1.c**) are incomplete.

Your task is to amend **ex1.c** and **rw_lock_struct.h** to solve the readers-writers problem. **`rw_lock`** should fulfill the following requirements:

- **Correctness:** The program is ensured to run correctly, according to the rules of the readers-writers problem as mentioned earlier.
- **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.

Only changes you made in **ex1.c** and **rw_lock_struct.h** will be used for grading. You may change the other files provided (**rw_lock.h**, **ex1_runner.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 and run use the following commands in **ex1** folder, respectively:

```
$ gcc -Wall ex1_runner.c ex1.c -o ex1 -lpthread
        //builds ex1 executable using 2 c files
        //-Wall shows all warnings
$ ./ex1 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.
```

2.2. Exercise 2

A problem with the basic solution:

Consider the following sequence of events that can occur:

```
1. Reader 1 requests access.
2. Reader 2 requests access.
3. Writer 1 requests access.
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** when such a sequence of actions occur. This can happen as long as one reader is accessing the shared resource before another reader requests access.

Exercise 2 Template

Exercise 2 main program **ex2_runner.c** is almost identical to **ex1_runner.c**. The only difference is that the **`reader`** threads are created before the **`writer`** threads.

You can compile and run **ex2.c** similarly to **ex1.c**. Likewise, there are also **rw_lock.h** and **rw_lock_struct.h** header files for exercise 2.

Your task for Exercise 2:

For this exercise, you can start from your solution in exercise 1. You *may* realize that using the same solution from exercise 1 would result in the following output:

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

This is because in the new program, readers would access the shared resource and "hog" them before any writers can do anything.

Your task is to amend the same files (**ex2.c**, **rw_lock_struct.h**), so that:

- All requirements from Exercise 1 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?)

Only changes you made in **ex2.c** and **rw_lock_struct.h** will be used for grading. You may change the other files provided (**rw_lock.h**, **ex1_runner.c**) during your own testing, but 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 3. Synchronization in a roundabout using semaphores.

For exercise 3, you may need to use general semaphores, that can be included with `<semaphore.h>`.

3.1 Exercise 3

In this exercise, you will design a traffic synchronizer to prevent crashes in a single-lane roundabout.

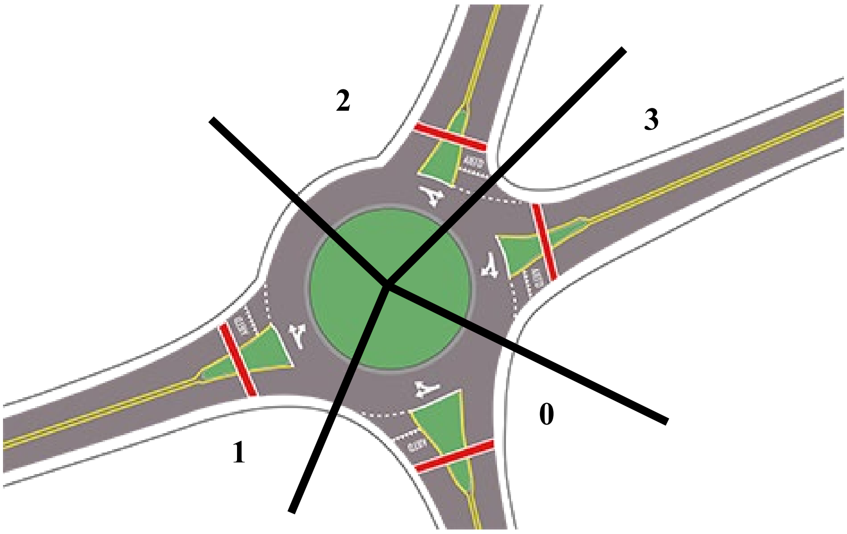

Figure 1: Roundabout with segments and streets

Figure 1 shows a roundabout with 4 streets connected to it. The roundabout is divided into multiple segments, each segment corresponding to a street. Considering that you have a roundabout with *n* streets, the segments and streets are numbered 0 to *n* - 1. 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, its next position would be 0, followed by 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).

To make sure that the **one car per segment rule** is adhered to, you must ensure that:

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).

[**No priority in the roundabout**] The 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!

Exercise 3 Template

ex3_runner.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 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` (implemented by you in file **ex3.c**).
4. The threads are joined.
5. Clean-up and printing are done.

ex3_runner.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.

ex3.c contains your code for function car, and additional initialization and clean-up if needed. Use **`initialize`** and **`cleanup`** functions to do any initialization you might need. They are called from **ex3_runner.c**.

Each car is modeled using a thread. The thread executes function **car**. A car should:

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.
5. You must ensure using synchronization mechanisms (semaphores) that you follow **one car per segment rule** with **no priority of cars in the roundabout**.

Compile all the files in **ex3/** folder and run:

```
$ gcc -Wall -DDEBUG ex3_runner.c ex3.c -o ex3 -lpthread
      //-DDEBUG flag is used to compile such that
      //debug information is printed. Remove the flag
      //if you do not want to see such information.
$ ./ex3 10000 5 20
```

ex3 takes in the following command line arguments:

- **`seed`** (e.g. 10000) to initialize the random number generator
- **`number_of_segments`** to specify how many segments to create
- **`cars_per_segment`** to specify how many cars to enter per street.

<u>Your task for Exercise 3:</u>

Your task is to amend **ex1.c** so that **`traffic_synchronizer for the roundabout`** can fulfil 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. (Get the first two requirements first.)

Only changes you made in **ex3.c** will be used for grading. You may change the other files provided (**traffic_synchronizer.h**, **ex3_runner.c**) during your own testing, but they will be replaced with the original files when we test your assignments.

Bonus (1%)

[**Priority in the roundabout rule**] In a real roundabout, a car moving in the roundabout has priority over a car waiting to enter from a street. Specifically, a car moving from segment *n-1* to segment *n* would have priority and should move to this segment (when available). Other cars waiting at street *n* to enter the roundabout would have to wait.

Your task for the Bonus grade would be to modify your answer from Exercise 3 in order to fulfil the **priority in the roundabout rule**, in addition to the other conditions. Please note that you need to submit both implementations:

- no priority in the roundabout [ex3 -> 3%]
- with priority in the roundabout rule [bonus -> 1%]

Some tips for Exercise 3

As mentioned in lecture, the three synchronization problems can be applied (with modifications) to solve many other synchronization problems – including this one!

First, identify which one this looks the closest to – if a car in segment n is moving, can cars in adjacent segments $n + 1$ and $n - 1$ move?

Secondly, what are the conditions that would cause a deadlock? In other words, under what conditions are cars all prevented from moving?

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):**

a. **`ex1/`**
   - **`ex1.c`**
   - **`rw_lock_struct.h`**
   - **`rw_lock.h`**
   - **`ex1_runner.c`**
b. **`ex2/`**
   - **`ex2.c`**
   - **`rw_lock_struct.h`**
   - **`rw_lock.h`**
   - **`ex2_runner.c`**
c. **`ex3/`**
   - **`ex3.c`**
   - **`traffic_synchronizer.h`**
   - **`ex3_runner.c`**
d. **`ex3-priority/`**
   - **`ex3.c`**
   - **`traffic_synchronizer.h`**
   - **`ex3_runner.c`**

Do **not** add additional folder structure during zipping, e.g. do not place the above in a "lab2/" subfolder etc.

Upload the zip file to the "Student Submission→Lab 3" workbin folder on LumiNUS. Note the deadline for the submission is **16th October, 2pm**.

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