THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 3

Due: 11:59pm Monday 2nd May (Week 9)

Task

This task will improve your programming skills dealing with dynamic arrays in C, parallel programming with pthreads, identifying race conditions and making critical sections thread-safe.

You have given an array of counters and threads. Each thread manipulates the counters by a given sequence of instructions. An instruction contains a reference to a counter, a work function that either increments, decrements, or doubles a counter, and the number of repetitions, i.e., how often an instruction is executed.

Data Structure

For this task you are given a data structure in C. You must use this data structure to solve your problem. The *only* alteration of the data structure which is permitted is to introduce mutices to make the data structures thread-safe.

```c
#ifndef __COUNTER_H__
#define __COUNTER_H__

#include <assert.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* counter data structure */
struct counter {
   long long counter;            /* to store counter */
};

/* counter value */
struct instruction {
   struct counter *counter;      /* pointer to counter */
   int repetitions;              /* number of repetitions  */
   void (*work_fn)(long long *); /* function pointer to work function */
};

#endif
```

The first data structure **counter** stores the state of the counters. You have two global variables to store the array of counters in your program:

```
/* number of counters  */
int ncounters;

/* counter array  */
struct counter *counters;
```

The first global variable stores the number of counters and the second global variable is a pointer to an array whose size is determined at runtime. This array needs to be dynamically allocated while reading the input of the program, and should be of size **ncounters**.

The second data structure **struct instruction** stores an instruction consisting of

1. a pointer to a counter that will be manipulated by the instruction,
2. the number of repetitions how often an instruction is executed,
3. a function pointer to a function that manipulates a counter.

Note that the work function specified in the data structure **struct instruction** manipulates directly the state of a **long long** counter rather than the data structure **struct counter**.

The instruction sequences for the threads are stored in a dynamically allocated two-dimensional array (**instructions**) with varying length per row. We use following variables, to maintain the array data structure:

```
/* number of threads */
int nthreads;

/* number of instructions  */
int *ninstructions;

/* array of arrays for storing instructions per pthread  */
struct instruction **instructions;
```

The first global variable **nthreads** stores the number of threads. The second variable **ninstructions** points to a dynamically allocated array of size **nthreads** containing the sequence lengths for each thread. Note that this array needs to be allocated dynamically. The last variable stores the two-dimensional array whose first dimension is the threads and the second dimension are the sequences whose length might vary. Technically, it is a pointer-pointer in C, i.e., **instructions** points to an array of pointers. This array contains for each thread an element pointing to an array of instructions.

To construct the two-dimensional array of varying row-length you need to allocate the array of pointers of length **nthreads** first and then for each pointer in this array you need to allocate a dynamic array of instructions whose size is determined by **ninstructions[thread]**.

Write a program that

1. reads in the input, allocate memory and constructs the dynamic data structure
2. spawns **nthreads** threads using **pthread_create**

3. let each thread execute its instruction sequence
4. join all threads in the main program,
5. display the state of the counter
6. free all dynamically allocated memory

The data structure as represented above is not thread-safe, i.e., two threads may manipulate the counter at the same time and may cause a race. Add a mutex to the data structure so that the operations become thread-safe. Make sure that you have a single mutex per counter[1].

Note that at the start of the program the counters are initialized to zero.

Input and Output

The structure of the input is as follows:

```
<number of counters>
<number of threads>
<instruction-sequence>
<instruction-sequence>
...
```

If there are n threads defined as `<number of threads>` in the input, you must have n occurrences of `<instruction-sequence>`.

An instruction sequence has following format:

```
<number of instructions>
<instruction>
<instruction>
...
```

If there are k instructions in the instruction sequence defined by `<number of instructions>` in the input, you must have k occurrences of `<instruction>`.

An instruction is defined as,

```
<counter> <work-function> <repetition>
```

where `<counter>` is the number of the counter (between `0` and `<number of counters>-1`), `<work-function>` is either `I`, `D`, or `2` standing for

I increment by one ($n' = n + 1$)

D decrement by one ($n' = n - 1$)

[1]Do not have a single mutex for the whole data structure – have a look at the lecture slides why not.

2 doubling the counter ($n' = 2n$)

and `<repetition>` represents the number of repetitions for this instruction, i.e., how often the work function is executed whilst executing the instruction.

For example, the input:

```
2
2
1
0 I 10
2
1 D 10
1 2 2
```

defines two counters and two threads. Because we have two threads, we have two instruction sequences. The first instruction sequence has length one, and contains a single instruction to increment the first counter with a repetition of 10. Note that counters are addressed starting from 0. The second instruction sequence has two instructions. The first decrements the value of the counter with 10 repetitions and the second doubles the value with two repetitions.

The output of the program is a sequence of numbers representing the state of the counters. For the example above we would obtain:

```
10
-40
```

In this example, the **`increment`** function *must* be invoked 10 times, the **`decrement`** function 10 times, and the **`mult2`** function 2 times. You should not expand out the calculation ahead of time. The testing framework will ensure that these functions are invoked the correct number of times.

If any errors are encountered during the reading of the input file format, your program should print out `error`, and then exit.

As with Assignment 2, your program will be checked for memory leaks. If your program leaks memory you will not pass the test case, even if the output is correct.

Writing your own test cases

Since your assignment will be automatically marked, it is crucial that you follow our instructions carefully. Your output will need to be in exactly the right format. To assist with this, we have made available some sample test cases. If you are logged in to the **`ucpu[01]`** machines, you can run these tests using the Makefile we provided by typing **`make test`**. If your output is incorrect, you will see both the expected output and your output. If you passed the test, then you will only see the name of the test.

The sample test cases are by no means exhaustive. You will need to test your code more thoroughly by thinking carefully about the specifications and writing your own tests.

As with the previous assignment, you can view the sample test cases manually. They are located in the folder **`~comp2129/assignment3/sample/question1`**.