School of Electrical Engineering, Computing and Mathematical Sciences
Curtin University

CMPE 4003/6004 – Concurrent Systems
Laboratory Assignment 2 of 3

DUE DATE: MON 12/10/2020 10.00 AM

This task involves parallel programming with POSIX threads (Pthreads).

For this you will need access to a C compiler and the associated Pthreads libraries.

It is *required* that you use Ubuntu 18.04.4 LTS to complete this laboratory assignment, as it already has the libraries for programming with Pthreads through the Native POSIX Threads Library (NPTL), thus there is no need to install additional packages.

You are required to complete the assessment tasks at your own pace, either in the lab on campus or at home, and then **submit a full report complete with source codes for grading.**

Ensure that you submit a *single* PDF of your report (filename SURNAME_STUDENTID_lab2report.pdf) plus a *separate* ZIP of all source codes generated (filename SURNAME_STUDENTID_lab2source.zip).

Follow the sequence of steps in each part by downloading the template codes for each part `lab2partX.c` and making the necessary modifications to the code as indicated in each question.

For each part the template has specified the structure for the code but certain sections of the code have been replaced with pseudocode comments indicating what should be executed in its place.

In preparing your lab report, ensure that where the problem:

- *requests an explanation, that it is given in sufficient detail, and as succinctly as possible,*
- *requests changes or additions to the code, to explicitly indicate the relevant line numbers as well as the modified or inserted code(s); DO NOT SIMPLY CUT AND PASTE YOUR ENTIRE SCRIPT*
- *requests execution of code, that your report shows the output, e.g. a screenshot or terminal session.*

P1. Recall the strategy for computing π using a Monte Carlo approximation. We sample uniformly points on a square dartboard. The centre of the square is at the origin and the sides are 2 units in length. The area of a circle which is inscribed just inside the square dartboard is then π square units. Now the ratio of the number of points falling in the square to those falling inside circle should approximately match the ratio of the areas of the circle and square respectively and thus *number in circle/number of tosses = π/4*.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number of tosses; toss++) {
      x = random double between - 1 and 1;
      y = random double between - 1 and 1;
      distance_squared = square of distance of dart from origin;
      if (dart toss falls in unit circle) increment number_in_circle
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

We will write a Pthreads program that uses a Monte Carlo method to estimate π. The main thread will fork multiple threads in order to parallelize the calculation. Global variables are used as shared variables between threads. Each thread carries out its own allocation of work and then adds its local count to a global count. We use `long long int` for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

(a) Briefly explain the basic difference between a process and a thread.
(b) Replace the commented pseudo code in line 45 with a single line of code so that iteration `i` of the loop forks a thread with attribute `thread_handles[i]` and executes the function `Thread_work` passing the thread rank `i` as an argument. Correspondingly replace the commented pseudo code of line 49 to join all the threads just created. Pay attention as to whether you need to pass in a pointer or the actual variable itself.
(c) Replace the comments in lines 78-79 in the code respectively by generating (uniformly distributed) random numbers `x` and `y` for the coordinates of the random dart tosses between -1 and +1. Now replace line 80 of the code to compute the distance of each dart from the centre origin and store the result in the variable `distance_squared`. Finally replace line 82 of the code to increment the local count if the darts fall within the unit circle. Ensure that your code is *thread safe*.
(d) Replace the comment pseudo code in lines 42 and 55 to initialize and destroy the mutex.
(e) Replace the comment pseudo code in lines 85,86,87 with 3 lines of C code to compute the global sum of all 'successful' samples ensuring that result is stored in the global variable `number_in_circle` and taking care to avoid a race condition.
(f) Compile and run your code with 10^9 tosses. Use 4 processes first. Then use 1.
Verify the results are correct. Is it faster with 4?

P2. We will write a Pthreads program that implements the trapezoidal rule. For ease of reference the integrand or the function to be integrated has been hard coded into the file. We will use a shared variable for the sum of all the threads' computations. For the critical section, we will consider three different approaches to enforcing mutual exclusion, namely mutexes, semaphores and busy-wait. We will use one code file that implements a switch to choose between the different mutual exclusion methods. For simplicity we will also assume that the number of trapezoids evenly divides the number of threads.

(a) Replace the commented pseudo code in line 73 with a single line of code so that iteration `i` of the loop forks a thread with attribute `thread_handles[i]` and executes the function `Thread_work` passing the thread rank `i` as an argument.
(b) Replace the commented pseudo code in line 78 with a single line to join the corresponding threads from the previous step.

Line 105 calculates a local trapezoidal rule for each thread's assigned interval. We need to then add the contribution of each thread to the global total sum:

(c) If each thread is allowed to increment the total sum on the fly, does a race condition exist? If so, how does this affect the final result?
(d) Replace the comment pseudo codes of lines 119,120,121 with three lines of code to implement a mutex based critical section that adds each thread's contribution to the total.
(e) Replace the comment pseudo codes of lines 109,110,111 with three lines of code to implement a semaphore based critical section that adds each thread's contribution to the total.
(f) Replace the comment pseudo codes of lines 114,115,116 with three lines of code to implement a busy-wait based critical section that adds each thread's contribution to the total.

Now the code is complete and ready to be executed.

(g) Compile and run your code to compute the trapezoidal rule for $a=0, b=1, n=2^{30}$. Use 128 threads and run for each of three methods coded for mutual exclusion. Verify that the result for each method is correct, checking by hand the true value of the integral. Which method is slowest/middle/fastest? Why?

P3. Now we will write a multithreaded file tokenizer which is thread safe. The basic idea is for each thread to read in consecutive but different lines of the file in a round robin fashion, and then for each thread to tokenize the words in the line which is assigned to the worker thread. Thus for our purposes the delimiter will be a blank space and the tokens are contiguous sequences of characters or simply individual words.

The main function first forks the threads, each runs the `Tokenize` function, and then the threads are joined. The `Tokenize` function is responsible for having each thread read individual and consecutive lines of the input file in a round robin fashion until the end of the file is reached, i.e. *using a cyclic distribution of the input lines to the worker threads*. After each line is read in, the thread to which it is assigned displays the line, and then calculates and displays the tokenized output. Thus if the number of input lines is much greater than the number of threads, and each line has roughly the same length, then each thread performs roughly the same amount of work as all of the other threads. In this case the efficiency of this multithreaded code is likely to be better with approximately equal distribution of the work amongst the threads.

The code has thread safe calls for string tokenizing and delimiter finding, and as supplied is ready to be compiled and tested.

(a) Run with 2 threads and the input file provided. Does the code run exhibiting the desired behaviour?
If yes, explain the flow of execution and how it achieves the desired solution, and do not proceed further.
If not, explain why not, and proceed below:
(b) Examine the main function and state the initial values of each semaphore for each thread or line.
(c) Consequently modify the `Tokenize` function to achieve the desired behaviour. Only modify the `Tokenize` function. Changes to other parts of the code are not permitted. Do not simply serialize the code by blocking threads, but ensure that your code tokenizes blocks of input lines in parallel.
Explain your solution.
Run your code with 2 threads and the input file provided to verify.

Hint: Use the semaphores to ensure that input lines are allocated to threads with a cyclic distribution and thus that no particular thread grabs more than its fair share of input lines. It is not necessary to ensure the output is displayed in consecutive order, only that the output for each thread is complete and correct.

P4. Consider the simple producer-consumer code provided in the template. The code is fully running and does not require any modification. Here producers are generating and writing information to the buffer while the consumers and reading and then deleting the information from the buffer. Read through the code and answer the following questions.

(a) What information is being written into and read from the buffer?
(b) Describe in detail the action of the producer and consumer thread as well as the flow of execution.
(c) Is there a race condition? Fully justify your answer.
(d) What is the producer thread doing while the consumer thread is active and vice-versa?
(e) What is the value in `buffer` just before the main function terminates and will it always be the same?

P5. Here we will investigate the impact of caching on parallel execution. We will use our own Pthreads version of a matrix vector multiplication. The code is complete with timing statements and there is no need to make any modifications. Browse through the code briefly so you understand the flow of execution.

(a) Compile and run the programme and complete the following table of run times and efficiencies.

Remember that the run time for a single trial is that of the slowest thread, and that we should benchmark the run time for our programme by taking multiple trials and picking the minimum.

	Matrix Dimensions					
	8000000x8		**8000x8000**		**8x8000000**	
Threads	Time (s)	Eff (%)	Time (s)	Eff (%)	Time (s)	Eff (%)
1						
2						
4						
8						

Now notice that for each of the matrix dimensions trialled, we have exactly 64000000 multiplications and additions, and so each case involves the same number of operations.

(b) Find out what processor and cache design your system is using, and in particular take note of the size of the cache line. Now discuss the results obtained, explaining why the efficiency varies as it does. Keep in mind that for each size of matrix tested, there is exactly the same number of multiplications and additions. What is the key factor that explains the difference in efficiency and especially in the 8x8000000 case?