School of Computing Sciences

**Module:** CMP-5013A Architectures and Operating Systems
**Assignment:** Threads and Thread Synchronisation

**Set by:** Gavin Cawley gcc@uea.ac.uk
**Date set:** 10th November 2020
**Value:** 10%

**Date due:** 23rd November 2020
**Returned by:** N/A
**Submission:** Bench inspection

### Learning outcomes

This coursework provides further experience of low-level systems programming in C, and the use of POSIX system calls, including error handling. This coursework focuses on the use of threads to exploit multi-processor and multi-core computer architectures and avoiding race conditions through the use of mutexes.

## Specification

### Overview

You are required to implement a multi-threaded program in C, using the POSIX pthreads library, to estimate the area of a unit circle by estimating the proportion of points in the smallest enclosing square that also lie within the circle. This is of course a *very* inefficient means of performing a very straightforward calculation, but it provides a simple example of how multi-core architectures can be exploited to improve performance of CPU intensive tasks.

You are strongly advised to watch the pre-recorded lectures "*OS2 - System Calls, Processes & Threads*" (slides) and "*Inter-process/Inter-thread Communication and Synchronisation*" (slides) that provide the background material for this coursework. The accompanying set of lab exercises "*OS1 - System Calls, Processes & Threads*" and "*OS2 - Threads and Thread Synchronisation*" provide useful information on the system calls required. You are also strongly advised to seek help from Dr Bostrom and the teaching assistants during the lab sessions for this coursework on Monday 16th November.

### Description

Figure 1 shows a unit circle centred at the origin (i.e. a circle with centre co-ordinates $(0,\ 0)$ with radius = 1), inscribed within the smallest possible enclosing square, with corners at $(+1,\ +1)$, $(+1,\ -1)$, $(-1,\ -1)$ and $(-1,\ +1)$. If we sample points distributed uniformly across the square, the proportion that will be also enclosed by the circle will be the ratio of the area of the circle and the area of the square. This means we can estimate the area of the unit square by sampling a large number of points (many thousands) from the square and finding the proportion of those that are also inside the circle and multiplying that by the area of the square (which is 4 square units).

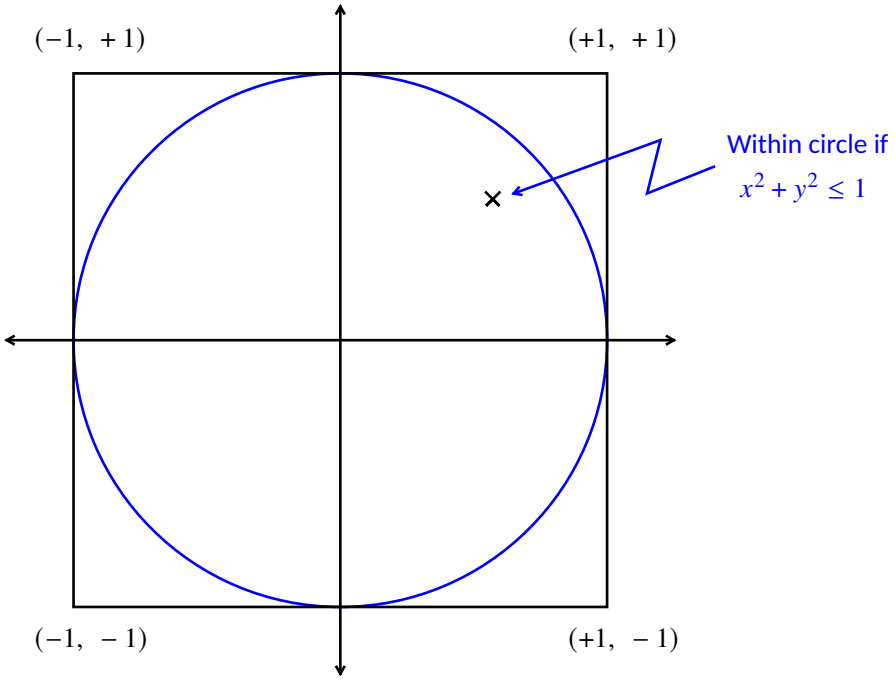


Figure 1: A unit circle inscribed withing the smallest enclosing square.

Of course this is *ludicrously* inefficient, you need many thousands (millions) of points to get a decent approximation of the area of the circle (which is of course [sic] just $\pi$ squared units), however the points can be generated, tested and counted in parallel, and so we can use threads to reduce the time taken. To simplify the task, it is split into three stages:

**Stage 1 -** Write a conventional single-threaded program that estimates the area of the unit circle by sampling a large number of points (specified as a command line argument, via `argc` and `argv`, so that it can be varied without recompiling the program). Make sure that it gives approximately the correct answer and that you fully understand the algorithm. The `stdlib.h` header includes a random number generator function, `rand` which generates random integer values between `0` and `RAND_MAX` (a symbolic constant defined in the header. This can be straightforwardly adapted to generate random floating-point numbers in the range $-1$ to $+1$.

**Stage 2 -** Write a multi-threaded version of the program, that generates a number of worker threads, each with its own work-space. Each worker thread performs its share of the computation separately and the results are pooled when all of the worker threads have completed. The work-space for each thread should record the number of points to be generated and the number of those points that were contained within the unit circle. Make sure you fully understand the operation of the `pthread_create` and `pthread_join` functions before attempting this exercise. Allow the user to specify the number of samples to perform and the number of worker threads to create, so that you can experiment with the effect of the number of worker threads on the total run time of the program (use the `time` command available on most UNIX systems).

Unfortunately the standard pseudo-random number generator, `rand`, is not thread safe; it has an internal seed variable, which determines the integer in the sequence. Happily `stdio.h` provides a thread safe version, `rand_r`, which expects a single parameter, a pointer to an `unsigned int` that can be used by `rand_r` to store the seed between calls (type `man rand_r` for further details).

**Stage 3 -** Adapt your solution from stage 2 so that the worker threads all share the same work-space, but to avoid a *race condition*, employ a *mutex* to ensure that only one thread can access the work-space at a time. Make sure you fully understand the operation of the `pthread_mutex_lock` and `pthread_mutex_unlock` functions before attempting this exercise. This will reduce the performance of the program quite considerably as the worker threads can no longer operate independently. What happens if you use a single, shared work-space but do not employ a *mutex*?

## Relationship to formative assessment

This coursework benefits from feedback provided on C programming skills provided by CMP-5015A Programming 2 (or comparable prerequisite modules).

## Deliverables

During the in-lab bench demonstration you will demonstrate that your programs function correctly (this may include directions from the marker to use particular command line arguments). You will also show your code to the marker and describe its operation and answer any questions the marker may pose.

## Resources

- The accompanying pre-recorded lectures "*OS2 - System Calls, Processes & Threads*" (slides) and "Inter-process/Inter-thread Communication and Synchronisation" (slides) and lab exercises "*OS1 - System Calls, Processes & Threads*" and "*OS2 - Threads and Thread Synchronisation*" provide useful background information.
- The `man` command on UNIX-like systems can be used to provide more detailed information on all available POSIX system calls.
- cppreference.com provides an excellent resource for information about C and the standard C libraries.
- The video "*An Introduction to Unix*" provides historical background on the UNIX operating system and an introduction to the Bash shell. The slides are available here, and the associated lab exercises (from week 1) here.

## Marking Scheme

Marks will be awarded according to the proportion of the specifications successfully implemented, programming style (indentation, good choice of identifiers, commenting etc.), and appropriate use of programming constructs. It is not sufficient that the program merely generates the correct output. Professional programmers are required to produce maintainable code that is easy for *others* to understand, easy to debug when (rather than "if") bug reports are received and easy to extend. The code needs to be well structured, and written so that it not only solves the problem specified today, but could easily be modified or extended to cater for future developments in the specification without undue cost. The code should be reasonably efficient (for the specified algorithm!). Marks may also be awarded for correct use of more advanced programming constructs or techniques not covered in the lectures. Students are expected to investigate the facilities provided by the POSIX library and the C standard libraries.

- Stage 1 - Single threaded program - 2 marks.
- Stage 2 - Multi-threaded program using separate workspaces - 3 marks.
- Stage 3 - Multi-threaded program using a single shared workspace - 1 mark.
- Quality of implementation (including error handling) - 2 marks.
- Explanation of programs and answers to questions - 2 marks.