Posix Threads - Exercises

Preliminary Note: before trying to solve these exercises, make sure you have understood and executed all examples from the theoretical slides.

PT1) (creating processes vs creating threads) Homework)

Execute the two programs of Appendix A of the PThreads slides deck in your Linux system and measure the execution time: **a)** enforcing the use of in a single-core; **b)** allowing several cores (if available) to be used. Try to explain the results observed.
Note: if `tasket` *is not available, you may install it in a Debian/Ubuntu system with the command* `sudo apt-get install util-linux`.

PT2) (chain of threads with old threads ending first)

Develop a program that involves **n** threads in addition to the **main** thread, as follows. Thread **main** creates the 1st thread of the chain and ends (printing a "bye!" message just before ending). In turn, the 1st thread creates a 2nd thread and ends (printing a "bye!" message just before ending). And so on, until the nth thread just ends (printing a "bye!" message just before ending), once it shouldn't create any new thread. All threads (including **main**) should print a "hi!" message as soon as they are born.

PT3) (chain of threads with newer threads ending first)

Develop a program that involves **n** threads in addition to the **main** thread, as explained next. Thread **main** creates the 1st thread of the chain, waits for its ending and also ends (printing a "bye!"message just before ending). Meanwhile, the 1st thread creates a 2nd thread, waits for its ending, and also ends (printing a "bye!"message just before ending). And so on, until the nth thread just ends (printing a "bye!" message just before ending), once it shouldn't create any new thread. All threads (including **main**) should print a "hi!" message as soon as they are born.

PT4) (pyramid of threads) (Homework)

Develop a program that involves a set of **n** threads (where **n** is a power of 2), including the **main** thread, as explained next. The thread **main** reads **n** from the user. If n>1, it creates a new thread; then, if n>2, both threads create a new thread each; then, if n>4, both threads create a new thread each; and so on. All threads (including **main**) should print an appropriate message when they are born (this message should show their pseudo-tid, and the depth at which they were created – **main** is at depth 0, the 1st new thread is a depth 1, the 2nd and 3rd new threads are at depth 2, and so on).
Note: before starting to write code, take some time to think on how to generate a unique pseudo-tid for each thread; knowing at which level each thread is at any moment is easy; making sure its pseudo-tid is unique is more challenging; remember that, at any moment, lots of threads may be in action and each one should define, independently of the others, the pseudo-tid of a new thread it is going to create.

PT5) (guess a random number)

Develop a program in which **n** worker threads (not including the **main** thread) try to guess a random number. This number is defined by the **main** thread before creating

the worker threads. As soon as one worker guesses the number, it should cancel all other workers. Meanwhile, the **main** thread should wait for the termination of all workers and should state which workers threads guessed the number; note that there may be several threads guessing the number and trying to cancel each other, at the some time; also, you may assume that the **main** will not be able to know about all threads that indeed guessed the number.
Note: be careful when choosing the way to generate random numbers generation in threads; be sure to select a thread-safe approach so that each thread will generate its own independent stream of randoms. Hint: see the man page of `rand_r`.

PT6) (approximate PI)

A way to approximate π, known as the *Monte Carlo method*, involves randomization:

- consider a circle of radius *r* inscribed in a square of side 2*r* (see Figure 1)

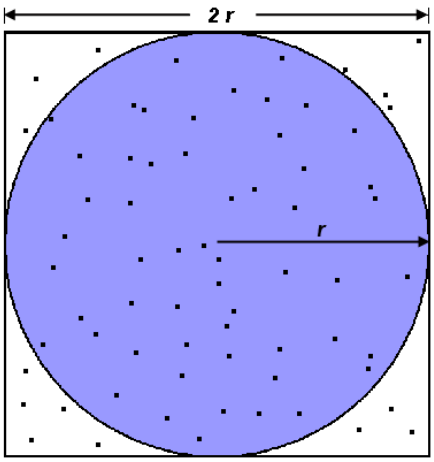

Figure 1 : A circle of radius *r* inscribed within a square of side *2r*.

- the area of the circle is πr^2 and the area of the square is $(2r)^2 = 4r^2$
- the ratio of the circle area to the square area is thus $\pi r^2 / 4r^2 = \pi / 4$
- generate **N** random points within the square, that is, generate **N** random pairs (x,y) such that $-r \leq x \leq +r$ and $-r \leq y \leq +r$
- a random point (x,y) falls within the circle if $\sqrt{x^2+y^2} \leqslant r$
- a proportion π / 4 of the **N** points will fall inside the circle and so the number of points inside the circle will be M = (π / 4) * N
- from the previous relation it follows that π ≈ 4 * M/N .

Develop a multi-threaded program that implements this algorithm for *r*=1 and **n** worker threads, where **n** matches the number of simultaneous threads supported by the CPU of your computer. Experiment the program for different values of **N** (overall number of random points) and confirm its influence in the quality of the approximation to π (consider π as given by the constant `M_PI` exposed by `math.h`).
Note: be careful when choosing the way to generate random numbers generation in threads; be sure to select a thread-safe approach so that each thread will generate its own independent stream of randoms. Hint: see the man page of `rand_r`.
Additional references:
https://en.wikipedia.org/wiki/Approximations_of_%CF%80
https://stackoverflow.com/questions/9912151/math-constant-pi-value-in-c

PT7) (parallel search in a vector - Take 1)

Develop a program to search for the minimum value of a vector of integers in parallel. The **main** thread should define a vector of N>=2^28 random integers (this takes >=1GB, once each integers takes 4 bytes). Then, T<=N worker threads should be used to search for the minimum. It will be up to the the main thread to show the minimum found. For a fixed value of N, execute the search with T=1,2,3,… and register in a worksheet the search time (for each value of T, execute the program 3 times and register the minimum time); stop increasing T when the search times start to increase instead of decreasing; draw graphs of execution times, speedup and efficiency, as a function of the number of threads. What conclusions can you take ?
Note: for very large arrays special GCC options may be needed; if you get some error from the compiler, use that error to search for a solution ...

PT8) (parallel search in a vector - Take 2)

In the previous exercise, you verified that increasing the number of threads does not yield the expected performance gains. In this exercise, you will search for a solution to that problem. **a)** start by identifying, in the serial version (T=1), the hot-spot(s) that consume most of the program execution time; this involves instrumenting the code to measure the duration of the most relevant code blocks (see *); **b)** apply Amdahl's Law to predict the realistic speedups achievable for the same values of T used in the previous exercise; compare the predicted speedups with the ones indeed achieved and take conclusions; **c)** see if it is possible to parallelize the hot-spot(s) identified and, if so, repeat the scalability study and verify the effect in the speedup and efficiency.
Hint (*): take a reading of https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9 and select a technique to measure the amount of time spent in the various sections of your code

PT9) (parallel search in a vector - Take 3) (Homework)

Revisit the scenario from the two previous exercises, with the following modifications: the vector should have the first N Fibonacci numbers; the goal is to find how many odd numbers exist in the vector. What conclusions can be taken concerning the possible speedup (as predicted by Amdahl's Law) brought from parallelization ? Compare the possible speedup with the real speedup achieved.

PT10) (parallel matrix multiplication)

Develop a program that implements the multiplication of two square matrices in parallel, by using **T** worker threads in addition to the *main* thread, where **T** matches the number of simultaneous threads supported by the CPU of your computer. For square matrices A and B of order n, the program should use **T** worker threads to cooperate in producing the matrix $C = A \times B$ of order n. Ensure that the load is evenly divided by the worker threads (use only values of n that divide equally by **T**). The initialization of the matrices A and B should be done using random numbers and also in parallel (ensuring thread safety and a different stream of randoms per worker thread). Add to your program a function that performs the multiplication in serial and another to compare two matrices. Use these functions to verify the correctness of the parallel product. Finally, measure the speedup achieved by the parallel multiplication.
Note: you can also use on-line tools, like https://onlinemathtools.com/matrix-multiply *or* https://matrix.reshish.com/ptBr/multCalculation.php *to verify the correctness of the results produced by your program.*