POSIX Threads Programming Exercises

Preliminary Notes:
- before trying to solve these exercises, make sure you have understood and executed the examples from the theoretical slides;
- in the exercises bellow, any function or primitive referenced by **`name(m)`** *means that you can access its manual page by issuing the command* `man m name` .

PT1 (creating processes vs creating threads) (homework)

Execute the two appendix_1 programs 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(1)` *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) (homework)

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.

PT2-1_fibonacci_recursive (Fibonacci variant of PT2) (homework)

Adapt the solution of program PT2 to generate the sequence of the first **2+n** Fibonacci numbers using **n** threads in addition to the **main** thread. The **main** thread should show the first 2 numbers of the sequence (0 and 1), the next thread of the chain should show the third Fibonacci number (0+1=2), the next thread of the chain should show the fourth Fibonacci number (1+2=3), and so on.

***Note**: the Fibonacci sequence is generated by the recurrence $F_n = F_{n-1}+F_{n-2}$, with $F_0=0$ and $F_1=1$ (see also https://en.wikipedia.org/wiki/Fibonacci_number).*

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

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 (*) (binary tree of threads) (homework)

Develop a program that involves **n** = $\mathbf{2}^{max\_level}$ threads, including the **main** thread, organized in a way that the creation of new threads follows a binary tree pattern: a thread at a certain ***level*** of the tree creates a new thread and both threads move to the next ***level***; this process starts with the **main** thread at ***level*** 0, and stops at each thread when its ***level*** reaches a predefined ***max_level***. Each thread should print a message stating its pseudo-tid (assume 0 for the **main** thread and 1,2, … **n-1** for the others), and the ***level*** at which it is currently (it may be a preexisting thread passing by that ***level*** of the tree, or it may be a thread just born at that ***level***). By the way, don't expect the output of the threads to appear grouped by level, once they progress and create new threads independently of each other – see the next example with ***max_level*** = 3:

```
thread 0 at level 0: hi!
thread 0 at level 1: hi!
thread 1 at level 1: hi!
thread 0 at level 2: hi!
thread 2 at level 2: hi!
thread 0 at level 3: hi!
thread 1 at level 2: hi!
thread 4 at level 3: hi!
thread 3 at level 2: hi!
thread 1 at level 3: hi!
thread 6 at level 3: hi!
thread 2 at level 3: hi!
thread 5 at level 3: hi!
thread 3 at level 3: hi!
thread 7 at level 3: hi!
```

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_collatz (check the Collatz conjecture for random numbers)

The Collatz conjecture [1,2,3] concerns what happens when given any positive integer p a next p is generated by applying the following algorithm: **if** p is even **then** $p=p/2$ **else** $p=3*p+1$ **fi**. The conjecture states that when this algorithm is continually applied all positive integers will eventually reach 1. For example, if initially $p = 35$, the algorithm generates the sequence 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

a) Code a serial program that implements the algorithm above for an initial random integer p and ensure it's running correctly. The program must stop when convergence to 1 is achieved (print the number of steps required to converge and then terminate) or an overflow happens (if $3*p+1 < p$, print "overflow" and then terminate).

Note 1*: use* `rand_r(3)` *to generate the initial p; this function requires a seed for the random number generator; to ensure a different initial p, ensure a different seed each time the program runs (e.g., use the process ID returned by* `getpid(2)` *as the seed).*

b) Implement a parallel version of the code developed in a) using Pthreads, with **w** worker threads in addition to the **main** thread, so that each worker thread verifies the Collatz conjecture for a random integer p_w, and the **main** thread just awaits for their termination. After convergence is achieved, each worker thread must show the number of steps required to converge and then terminates. If there is an overflow in a worker thread, it should print "overflow" and then terminate its execution.

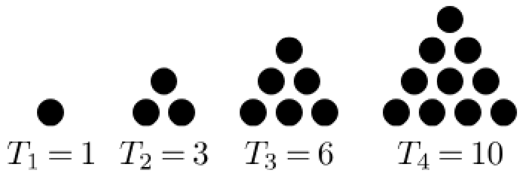
***Note 2**: in each worker, call* `rand_r(3)` *to generate the initial p, using as seed the unique opaque id of the worker as returned by* `pthread_self(3)`, *once that opaque id will be different for every thread and will be different every time the program runs.*

***References**:*

[1] *https://www.youtube.com/watch?v=094y1Z2wpJg*
[2] *https://en.wikipedia.org/wiki/Collatz_conjecture*
[3] https://www.quantamagazine.org/why-mathematicians-still-cant-solve-the-collatz-conjecture-20200922/

PT6_triangular_numbers (parallel generation of triangular numbers)

A triangular number counts objects arranged in an equilateral triangle: the *n*th triangular number is the number of dots in the triangular arrangement with *n* dots on each side. For instance, the first four triangular numbers (1,3,6,10) are represented by:



The *n*th triangular number is directly given by the formula $T_n = n(n+1)/2$. Implement a Pthreads-based parallel program to generate the first **N** triangular numbers by applying the previous formula. Use **w** worker threads in addition to the **main** thread. The parallel solution must spread the load as uniformly as possible by the **w** workers (set **w** = number of CPU cores of your machine). Have the **main** thread to define the work slices of each worker (and do not assume **N** to be evenly divisible by **w**). Store the **N** triangular numbers in a global vector and have the **main** thread to show it before ending the program. Confirm the 1st 100 numbers match those available in [1].

***References**:*

[1] *https://en.wikipedia.org/wiki/Triangular_number*

PT7_fibonacci_parallel (Fibonacci sequence generated in parallel)

The Fibonacci sequence is generated by the recurrence $F_n = F_{n-1}+F_{n-2}$ with $F_0=0$ and $F_1=1$. This recurrence cannot be computed in parallel, due to the dependencies involved. However, using Binet's formula [1], F_n is directly computable by rounding (to the nearest integer) the expression $B_n = \varphi^n / \text{sqrt}(5)$, where $\varphi = (1 + \text{sqrt}(5)) / 2$.

a) Code a serial program that generates the first **N** Fibonacci numbers using the recurrence $F_n = F_{n-1}+F_{n-2}$ and Binet's formula. Stop generating the numbers if there is an overflow ($F_n < F_{n-1}$) or mismatch ($F_n \neq B_n$). Print both numbers, side-by-side:

```
n       Recurrence    Binet
0       0             0
1       1             1
2       1             1
3       2             2
4       3             3
...
```

***Note**: the Recurrence and Binet's numbers will match only up to a certain point, after which Binet's numbers diverge due to floating-point representation errors.*

b) Implement a Pthreads-based parallel program to generate the first **N** Fibonacci numbers by applying the Binet's formula (ignore any overflows). Use **w** worker threads in addition to the **main** thread. The parallel solution must spread the load as uniformly as possible by the **w** workers (set **w** = number of CPU cores of your machine). Have the **main** thread to define the work slices of each worker (and do not assume **N** to be evenly divisible by **w**). Store the **N** Fibonacci numbers in a global vector, in ascending order, and show it. Confirm results match those produced in a).

c) Implement a variant of b) where the main thread is also a worker thread.

d) Implement a variant of c) where **N** is assumed to be divisible by **w** (so that every worker can easily define its own work slice, not needing the **main** thread to define it).

References:

[1] https://en.wikipedia.org/wiki/Fibonacci_sequence#Binet's_formula

PT8_leibnizPI (approximate PI via a simple sum)

One way to get a numerical approximation to π is through the formula

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

This is known as the Gregory–Leibniz series [1], which in serial code is simply:

```
    unsigned long k, T=...;
    double factor = 1.0;
    double sum = 0.0;
    for (k=0; k<T; k++) {
         sum += factor / (2*k+1);
         factor = -factor;
    }
    pi_approx = 4.0*sum ;
```

where `T` is the number of consecutive terms of the form $(-1^t)*1/(2t+1)$, t=0,1… to use.

a) Start by coding a program with the serial version above and make sure it is running correctly. It is up to you to decide the value of `T`, but consider that you probably need a big value in order to reach a fair approximation to π. Run the program with different values of `T` and verify its influence in the quality of the approximation to π. Consider the "true" value of π as the one of the `double` constant `M_PI` exposed by `math.h` [2] and compare it to your approximation to calculate the representation error.

b) Implement a parallel version of the code developed in a) using Pthreads, with **w** worker threads (including the **main** thread). The parallel solution must spread the load (partial sums) as uniformly possible by the **w** workers. Store the partial sums in a global vector, where each cell saves the partial sum produced by each worker thread, and that cell is updated as the partial sum is being calculated by the respective worker. Before the program ends, the **main** thread will show the approximation achieved to π.

c) Comparing the execution times of versions a) and b) reveals that the parallel version b) is slower than the serial version a). The underlying problem, harming performance, is known as ***false sharing*** [3] and here manifests in the following way: the global vector of **w** partial sums consumes a certain number of memory blocks; each memory block stores a certain number **w'** <= **w** of partial sums; for a memory block to be changed, it must be copied to the CPU cache, to a certain cache line; for **w'** worker threads to update **w'** partial sums in the same block (one partial sum per thread), there must be **w'** copies of that memory block, one copy in the cache of the CPU running each thread; now, every time one of the **w'** threads updates its specific partial sum, in its own cache line, all remaining **w'-1** copies of the same cache line, in the other CPU cores, must also be updated, per the requirements of the cache coherency protocol (and despite the fact that different threads are never updating the same partial sum); so, the **w'** threads are hit by a performance penalty from a ***false sharing*** situation. To minimize or solve the problem, adopt the following strategies.

c1) Produce a variant of b) where a local variable in each thread accumulates the thread's partial sum and the thread's cell in the global vector is only updated with the final value of the thread's partial sum. Copy also to thread local variables the coordinates of the thread specific slice (begin and end of its partial sum).

c2) Check the size of the cache line for all cache levels of your CPU. For instance:

```
$ ls -la /sys/devices/system/cpu/cpu0/cache/
... index0
... index1
... index2
... index3

$ cat /sys/devices/system/cpu/cpu0/cache/index[0-3]/coherency_line_size
64
64
64
64
```

In this case, to make sure different threads never access the same cache line, each thread cell in the global vector of partial sums must consume a full cache line. **Hint**: consider using structures for the vector cells, adding a padding field to the partial sum, in order to ensure the total size of the structure matches the cache line size (64 bytes).

You should find the c1) and c2) times similar between them, and now faster than a).

Note*: you may notice small differences between the approximation to π generated by the different code versions; it is expected and due to rounding errors when summing doubles by a different relative order; to understand this, consider how the terms are added in the serial version vs how they are added in the parallel versions ...*

References:

[1] https://en.wikipedia.org/wiki/Approximations_of_%CF%80
[2] https://stackoverflow.com/questions/9912151/math-constant-pi-value-in-c
[3] https://en.wikipedia.org/wiki/False_sharing

PT9_randPI (approximate PI via randomization) (homework)

A known way to approximate π involves randomization (aka Monte Carlo method):

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

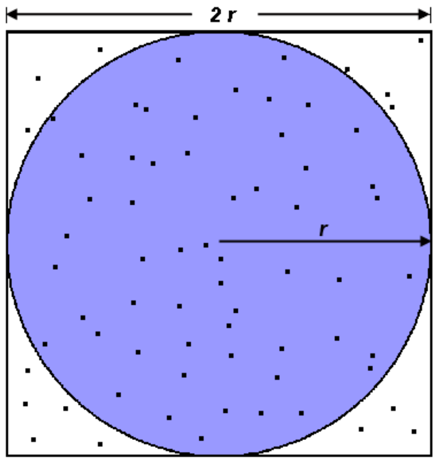


Figure 1 : A circle of radius *r* inscribed within a square of side *2r*.

- 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 **A** random points within the square, that is, generate **A** 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 **A** points will fall inside the circle and so the number of points inside the circle will be **B** = (π / 4) * **A**
- from the previous relation it follows that π ≈ 4 * **B**/**A** .

Develop a multi-threaded program that implements this algorithm for *r*=1 and **w** worker threads (set **w** = number of CPU cores of your machine). Experiment the program for different values of **A** (overall number of random points) and confirm its influence in the quality of the approximation to π (as given by the constant `M_PI`).

Notes:
- apply the same thread-safe approach already used in exercise PT5_collatz in order to ensure each thread will generate its own independent stream of randoms.
- remember that each random is originally an integer in the range [0, `RAND_MAX`*] and so you need a extra step to convert any random to the range [0, 1] ...*

PT10 (other approximations to PI) (homework)

There are many algorithms to approximate π with repeated calculations that may be easily parallelized. Many (including those already explored in the previous exercises) may be found in https://en.wikipedia.org/wiki/Approximations_of_%CF%80. Go to this web page, select a different algorithm and apply the same approach as in exercise PT6.

PT11 (competition to guess a random number)

Develop a multi-threaded program in which **w** 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 a 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 that "simultaneously" guess the number, cancel all others, and are still able to "tell" **main** that they guessed; also, you may assume that the **main** may not be able to know about all threads that indeed guessed the number).

Notes:
- apply the same thread-safe approach already used in exercise PT5_collatz in order to ensure each thread will generate its own independent stream of randoms.
*- if you find that the worker threads take a lot of time to guess the random number generated by the **main** thread, shrink the range of the randoms*
*- consider that a worker thread may be very fast in guessing the number and when trying to cancel the co-workers some may have not yet been created by **main** !*
- consider using `pthread_testcancel` *to improve the chance of cancellation*

PT12 (parallel access to a vector - take 1)

a) Develop a serial program to determine how many odd numbers are there in a sequence of random numbers (note that if the random generator ensures a uniform distribution, the amount of odds and evens should be the same). Start by filling a global vector of **n**=2^28 cells of type `unsigned int`, with random values; enclose the code for this task in a function `init_vector`. Then, count how many odd numbers are there in the vector; enclose this task in a function `count_odds` that returns the number of odds found. Finally, print this value and end. Note that this program may generate different results for each run, depending on how the seed of the generator is defined.

b) The initial seed of a pseudo-random number generator defines the entire sequence of randoms that will be generated, meaning the sequence is in fact deterministic (only apparently random). Once the next number is a function of the current internal state of the generator, the generation of a specific random sequence is intrinsically sequential, meaning the function `init_vector` is not parallelizable. However, `count_odds` is clearly parallelizable, and it is possible to predict the theoretical speedup from its parallelization by applying Amdahl's Law. This requires measuring the fraction of the execution time of the serial program spent in the `count_odds` function (see next).

b1) Instrument the code from **a)** to measure the execution time of the full program and the execution time of the `count_odds` function; measure these times in nano-seconds [1] and show them just before the program ends. To collect these measurements, do 4 runs of the program, and register the 3 smallest values (k-best approach, with k=3) in the companion spreadsheet to this exercise (if necessary, repeat the procedure until the relative standard deviation of the selected times is less than 10% of their averages).

b2) Another way to determine the fraction of the time spent by the serial program in the `count_odds` function – and, in fact, in any other of its functions – is through a profiling tool. One possibility is to use the `gprofng` tool along with its GUI interface **[2,3]**. Install both (`sudo apt install binutils gprofng-gui`) and then execute:

```
gprofng collect app ./exercisePT12a.exe
gprofng display text -functions test.1.er
gprofng display text -calltree test.1.er
gprofng display gui test.1.er
```

Confirm that the fraction of the time spent in the `count_odds` function is similar to the one measured by the instrumented version of the program (**b1)**).

c) Based on the results of **b1)**, use the same spreadsheet to apply Amdahl's Law in order to predict the theoretical (ideal) values of the speedup, efficiency and execution times of a parallel version of program **a)** when running with 2 to 8 CPU-cores.

d) Develop a multi-threaded version of program **a)** with a focus on the parallelization of the `count_odds` function. Adopt the good practices from previous exercises (the main thread should also be a worker, the workload should be spread uniformly by all workers, false sharing should be avoided or minimized). Measure the overall execution time with the same instrumentation code used in b1). For each number of workers, varying from 2 to 8, run the program 4 times and register the best 3 times in the companion spreadsheet. The real speedups and efficiencies, and their ratios to the ideal ones, will be automatically calculated and plotted. Finally, draw conclusions.

Notes:
- before compiling the various codes of question PT12, make sure the `Makefile` within the source code folder has the definitions `OPT=-O0` and `DEBUG=-g` active.
- to run these programs with global arrays larger than 1GB, they must be compiled with the `gcc` option `-mcmodel=medium`; the `Makefile` supplied with the source code of the solution already caters for this; for more details see [4,5]

References:
[1] https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9
[2] https://sourceware.org/binutils/docs-2.40/gprofng.html
[3] https://blogs.oracle.com/linux/post/the-gprofng-gui
[4] https://stackoverflow.com/questions/10486116/what-does-this-gcc-error-relocation-truncated-to-fit-mean
[5] https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/x86-Options.html#index-mcmodel_003dmedium-1

PT13 (parallel access to a vector - take 2) (homework)

Revisit the scenario of exercise PT12 and replace the function `count_odds` by `find_min`. The goal is now to find the smallest value of the global vector of randoms.

PT14 (parallel dot product)

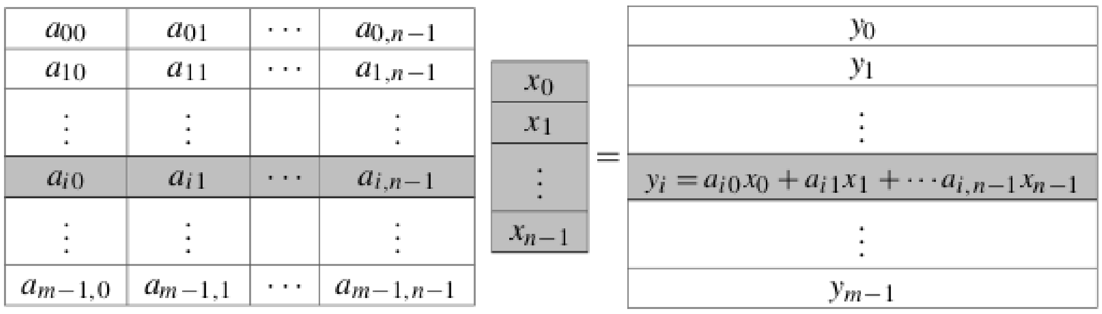
Develop a multi-threaded parallel program to perform the dot product of two vectors **a** and **b**, with **n**=2^28 integers each (to simplify the scenario, use fixed integers, e.g., 1's for **a** and 2's for **b**) by using **w** worker threads in addition to the **main** thread. The **main** thread should print the value of the dot product. Spread the load as uniformly as possible by the **w** workers for both these situations: **n>=w** (there is enough work for all workers); **n<w** (one or more worker threads will have no work assigned). Also, assume the initialization of the vectors **a** and **b** to be done serially by the **main** thread.

***Note**: consider the algebraic definition of the dot product given in https://en.wikipedia.org/wiki/Dot_product*

PT15 (parallel matrix - vector product) (homework)

Consider the product of a **mxn** matrix **A = (a_{ij})** by a **n**-dimensional column vector **x = (x_0, x_1, ..., x_{n-1})^T**. The result of this product is **y = Ax**, a **m**-dimensional column vector **y = (y_0 , y_1, ... , y_{m-1})^T**, in which the **i**'th component, **y_i** , is obtained by finding the dot product of the **i**'th line of **A** by **x** (see the next formula and companion figure).

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j .$$



Develop a multi-threaded parallel program to solve this problem. To keep the scenario simple, initialize **x** with 1's, line 0 of **A** with 1's, line 1 of **A** with 2's, and so on (i.e., the i'th line of **A** is initialized with i+1). Assume the initialization of **x** and **A** to be done serially by the **main** thread. However, the product **Ax** must be done in parallel; this should be performed by **w** worker threads ensuring an uniform load among them. To make the program user friendly, code auxiliary functions to show the content of **A**, **x** and **y** (to invoke after the initialization of **A** and **x**, and after **y** has been computed).

***Hint**: how will you divide the work ? the picture above hints at one possibility ...*

***Note**: to develop and test your solution, it is enough to use small values of* **m** *and* **n** *(e.g.,* **m**=**n**=*10); in such case it would be viable to initialize* **A** *and* **y** *in their declaration; however, to keep the solution generic, always initialize via explicit code.*

PT16 (parallel matrix - matrix product for square matrices) (homework)

Consider the product of a **nxn** matrix **A = (a_{ij})** by a **nxn** matrix **B = (b_{ij})** - these are also called square matrices of order **n**. The result of this product is **C = AB = (c_{ij})**, a matrix of order **n** where the component **c_{ij}** is obtained by computing the dot product of the **i**'th line of **A** with the **j**'th line of **B** (see the next figure for an example with **n**=3).

$$\begin{Bmatrix} 8 & 2 & 9 \\ 2 & 8 & 1 \\ 2 & 8 & 7 \end{Bmatrix} \begin{Bmatrix} 2 & 5 & 7 \\ 6 & 7 & 2 \\ 3 & 5 & 7 \end{Bmatrix} = \begin{Bmatrix} 8\times2+2\times6+9\times3 & 8\times5+2\times7+9\times5 & 8\times7+2\times2+9\times7 \\ 2\times2+8\times6+1\times3 & 2\times5+8\times7+1\times5 & 2\times7+8\times2+1\times7 \\ 2\times2+8\times6+7\times3 & 2\times5+8\times7+7\times5 & 2\times7+8\times2+7\times7 \end{Bmatrix}$$

Figure 3 : Square matrix multiplication – example with real numbers for **n=3**

a) Develop a multi-threaded parallel program to solve this problem. To simplify the scenario, initialize line 0 of **A** and **B** with 1's, line 1 of **A** and **B** with 2's, and so on (i.e., initialize the i'th line of **A** and **B** with i+1). The initialization of matrices **A** and **B** should be done serially by the **main** thread, but their product should be performed in parallel by **w** worker threads in a way that ensures an uniform load distribution (assume **n** is large enough to keep all **w** workers busy). Also, develop one auxiliary function to show the content of a matrix, and another to perform the matrix product in serial and compare the result with the outcome of the parallel product; (*) before the program ends, the thread **main** should show **A**, **B** and **C**, and confirm **C** to be correct.

b) Change the solution of **a)** to also initialize **A** and **B** in parallel. Note that you must use some form of global synchronization to ensure all worker threads have initialized their "slices" of these matrices before they all move to the computation of the product. Caveat: solve this problem without resorting to mutexes neither condition variables.

c) Change the solution of **b)** to initialize **A** and **B** in parallel with floating point numbers, either **c1)** simple-precision (`float` C type) or **c2)** double-precision (`double` C type). You may use the same numbers of **b)**, but converted to floating-point format.

d) For the same **n** (e.g., **n**=1000), compare the performance (execution times) of versions **a)**, **b)**, **c1)** and **c2)** with different values of **w** (at least 1,2,4,8) and draw conclusions. For this comparison, do not execute the code (*) in the **main** function.

e) Consider a variant of **b)** in which **two crews of worker threads** are used, one for the initialization phase, another for the multiplication phase; this means the separation between the two phases is simply achieved by having the **main** thread joining with the first crew and then having to create the second crew; compare the execution time of this new approach with that of **b)** for **n**=1000 and **w**=8 threads per crew.

Hint: *how will you divide the work ? think on the approach used in exercise PT13 and see if it is somehow reusable/adaptable; do you see any other possibilities ?*
Note 1: *to develop and test your solutions, start with small values of* **n** *(e.g.,* **n***=10); if the solutions seems robust, then test with much larger matrices*
Note 2: *note that instead of initializing* ***A*** *and* ***B*** *with deterministic numbers, you could use random numbers and this initialization is also easily parallelizable by applying the same technique used in* ***b)*** *(which is basically "global active waiting")*
Note 3: *you can use on-line tools to check the the correctness of the results; see:*
- *https://onlinemathtools.com/matrix-multiply or https://matrix.reshish.com/ptBr/multCalculation.php*

Note 4: *you can simplify the declaration of 2D matrices as fixed length arrays; see:*
- https://stackoverflow.com/questions/4523497/typedef-fixed-length-array

PT17 (*) (parallel matrix - matrix product: generalization) (homework)

Revisit exercise PT16a and develop a general solution for the multiplication of matrices of any order, that is, a solution that calculates $C_{m\,x\,p} = A_{m\,x\,n} \times B_{n\,x\,p}$ in parallel.

PT18 (different alternatives to joining)

Code a program that creates **w** worker threads, each worker prints its pseudo-tid and ends. The **main** thread also prints a message, but only after all workers have printed their messages. Instead of using passive waiting with `pthread_join`, ensure this behavior using **a)** busy waiting without mutexes; **b)** busy waiting with mutexes; **c)** passive waiting with condition variables; **d)** passive waiting using barriers.

PT19 (approximate PI via a simple sum - take 2)
(protecting shared global variables with mutexes)

Revisit exercise **PT8c1**. It involves a situation in which a global vector is being used to deposit/share the contributions of worker threads, with a specific cell per thread. Adapt the code to take advantage of mutex variables. Explore two approaches: **a)** *coarse-grain*: a variable shared by many threads is minimally accessed by each thread (e.g., to account only for the local contribution of the thread); **b)** *fine-grain*: a variable shared by many threads is intensively accessed by each thread (e.g., inside a loop). Compare the execution time of the three approaches (PT8c1, PT18a, PT18b).
Note*: you can apply the same techniques in exercises PT9, PT10, and PT12 to PT17.*

PT20 (competition to guess a random number - take 2) (homework)
(protecting shared global variables with mutexes)

Recall exercise **PT11** and consider the following variant: as soon as a thread guesses the number, it should "register" that fact and exit; threads that are still trying to guess should check if someone else guessed, in which case they should exit; thread **main** awaits for the death of all threads and will be able to tell which threads guessed (note that may be more than one thread that may have guessed at the same time). Implement this variant using mutex variables to protect access to variables shared by all threads.

PT21 (compute average distance between random points in a square)
(protecting shared global variables with mutexes)

Consider a square of side *a*, and *r* random points inside the square. It can be proved [1] that the average distance between those *r* points is ≈ 0.52140...**a*. To confirm this by brute-force, one can calculate the distance *d(r1,r2)=sqrt((x1-x2)^2+(y1-y2)^2)* between any pair of random points *r1=(x1,y1)* and *r2=(x2,y2)*, sum all those distances and divide the result by *r(r-1)/2* (which is the number of pairs of random points).

a) Code a serial program that validates the formula 0.52140...**a* for *a=1* and *r=1000000*; assume the bottom left corner of the square to be the point *(0,0)*, meaning any random *x* and any random *y* must fall between 0 and 1.

b) Develop a multi-threaded variant of a), where the work involved in proving the formula is divided, as uniformly as possible, by **w** threads (including the **main**); note that the usual slice-based approach is not directly applicable: *r* distances must be calculated between the 1st random point and the remaining *r-1* points, *r-1* distances must be calculated between the 2nd random point and the remaining *r-2* points, etc.; this means that the amount of work diminishes as we move in the sequence of random points; therefore, how can one still ensure that all threads have a similar workload ? ***Hin***t: use a *work-queue* approach, in which an index over the sequence of random points is used to assign a unit of work to a thread - for a current index *i* between *0* and *r-1*, the next thread asking for work will be assigned the task of calculating the distances between point *i* and all points *i+1* to *r-1*, also ensuring that *i* is atomically incremented to *i+1* so that the next thread starts one position ahead; this will ensure load balancing, because the fastest thread will receive the smallest *i* available, which marks the beginning of the largest subsequence of random pairs yet to be processed.

References:

[1] *https://mindyourdecisions.com/blog/2016/07/03/distance-between-two-random-points-in-a-square-sunday-puzzle/*

PT22 (finding the 1st N prime numbers) (homework)
(protecting shared global variables with mutexes)

Develop a program to find the 1st **N** prime numbers in parallel, starting at number 0. The primality test for a number **n**>= 0 must use the following function (see [1]):

```
char isPrime(unsigned long n)
{
  if (n == 2 || n == 3)  return 1;
  if (n <= 1 || n % 2 == 0 || n % 3 == 0) return 0;
  for (unsigned long i = 5; i * i <= n; i += 6)
      if (n % i == 0 || n % (i + 2) == 0) return 0;
  return 1;
}
```

The search should be done by **w** worker threads using a *work queue* model: a global variable **n** holds the next candidate to be tested for primality; a worker thread should pick up a copy of the current value of **n** to test it; that worker should also increment **n** one unit, so that **n+1** is the next value to be tested for primality; if a worker finds that the number it tested is prime, it should add that number to a global vector **v** (a vector with **N** cells, where the **N** primes found are to be stored); if a worker finds that **v** is full, it should stop searching for more primes and exit; before ending, each worker should present the quantity of candidates it tested, and the quantity of primes found among those candidates. Because worker threads may insert primes in **v** by any order, **v** may not be sorted; the **main** thread should sort **v** using the `qsort(3)` GLIBC function and then show it (one prime per line). To validate the output, compare it with the lists of primes available at [2] or generated by `sieve2310_64bit` [3].

References:

[1] https://en.wikipedia.org/wiki/Primality_test
[2] https://simple.wikipedia.org/wiki/List_of_prime_numbers
[3] https://www.rsok.com/~jrm/source/

PT23 (access to a "database" using mutex variables and/or read-write locks)

a) Implement the following scenario using mutex variables and read-write locks, for synchronization, where appropriate: a **main** thread initializes a vector **v** (a "database") of **c** cells ("registers") with 0's; it then creates **w** worker threads to perform an overall number **t** of transactions (each thread will perform a part of these transactions, but the number of transactions performed by each thread is inherently unpredictable); worker threads are alive while the number of transactions already performed is less than **t**; for each transaction, a worker must randomly decide if it is a read-transaction or a write-transaction, and it must do that choice in a way that ensures that only a percentage **p** are write-transactions; the index **i** (in 0..**c**-1) of the cell accessed in a transaction must also be randomly chosen; a read-transaction just makes a copy of v[i] to a local variable of the worker; a write-transaction changes v[i] to 1; after all workers have finished, the **main** thread will output the percentage of the vector **v** that was effectively changed to 1. Consider **t**=10000000, **c**=2000000, **p**=2% and **w** accordingly with the number of hardware threads supported by the CPU of your machine.

b) Change the solution of a) in order to use only mutex variables. Compare the execution time of both solutions for different values of **p**. Draw conclusions.

c) (*) (homework) Implement a variant where **v** is dynamic and `realloc(3)` is used to grow the vector once cell at a time; initially, **v** is empty (**c**=0) and so the first transaction must be a write-transaction (v[0]=1); afterwards, the kind of transactions may start to be randomly chosen, accordingly with the parameter **p**; for every write-transaction it must be randomly decided if an existing cell of **v** is to be changed (set to 1), or a new one must be added to **v** and initialized (set to 0).
Hint: instead of growing one cell at a time, consider growing a chunk of cells at a time (this makes `realloc` less intrusive, but will add extra complexity to the solution)

d) (*) (homework) Investigate the `tsearch(3)` family of GLIBC functions, that implement a binary search tree, and change the solution of c) to replace the dynamic vector with a binary search tree. Compare the implementation complexity, the performance and the amount of memory used by both approaches. Draw conclusions.

PT24 (threads alternating execution using condition variables)

Develop a program with two threads: a *count* thread and an *output* thread (one may be the **main** thread). The *count* thread should increment an `unsigned long long` counter indefinitely, one unit at a time; whenever the counter becomes a multiple of 1 000 000, the *output* thread should print the counter before the *count* thread continues.
Hint*: this scenario is very similar to that of example13B ...*

PT25 (competition to guess a random number – take 3) (homework)
(synchronization based on condition variables)

Revisit the scenario of exercise **PT11** and develop a solution without thread cancellation based on condition variables: the first worker that guesses the number should "notify" the **main** thread about it and should provision for the other threads to stop guessing. Also, the **main** thread should be able to tell which worker(s) guessed.

Output Example with 4 threads:	Output Example with 4 threads:
thread main: number to guess is 451199684 thread 2: guessed after 49412941 tries thread 2: I am the first to guess thread main: thread 2 guessed thread main: program completed: exiting thread 1: thread ?? already guessed: exiting thread 0: thread ?? already guessed: exiting thread 3: thread ?? already guessed: exiting	thread main: number to guess is 451199684 thread 0: guessed after 100 tries thread 3: guessed after 500 tries thread 0: I am the first to guess thread 3: thread 0 guessed before me thread 1: thread ?? already guessed: exiting thread 2: thread ?? already guessed: exiting thread main: thread 0 guessed thread main: program completed: exiting

PT26 (different approaches to implementing/using barriers)

Code a program with **w** worker threads that goes through 3 stages in synchrony: in stage 1, all workers print their pseudo-tid and "A"; after all workers finish stage 1, all move to stage 2, to print their pseudo-tid and "B"; after all workers finish stage 2, all move to stage 3, to print their pseudo-tid and "C", and then end. The **main** thread may die after creating the workers. Explore and compare the following synchronization approaches: **a)** flags; **b)** mutexes; **c)** condition variables; **d)** barriers.

PT27 (access to a "database" - take 2) (homework)
(synchronization based on condition variables or barriers)

Revisit exercise **PT23** and create an extra worker thread to output the percentage of the vector **v** that was effectively changed to 1, instead of that task being performed by the **main** thread. Solve the problem using: **a)** condition variables; **b)** barriers.

PT28 (*) (binary tree of threads - take 2) (homework)
(synchronization with barriers)

Revisit the scenario of exercise **PT4**. Make the necessary changes to the code of its solution to ensure that all threads passing by or born at a certain ***level*** all meet there before moving on to the next ***level***. Note that such requirement will ensure the output of the threads will now appear grouped by ***level***. For instance, for the same example given in exercise PT4, we would now have something like:

```
thread 0 at level 0: hi!
thread 0 at level 1: hi!
thread 1 at level 1: hi!
thread 0 at level 2: hi!
thread 2 at level 2: hi!
thread 1 at level 2: hi!
thread 3 at level 2: hi!
thread 0 at level 3: hi!
thread 4 at level 3: hi!
thread 1 at level 3: hi!
thread 6 at level 3: hi!
thread 2 at level 3: hi!
thread 5 at level 3: hi!
thread 3 at level 3: hi!
thread 7 at level 3: hi!
```