# Laboratory #3: **Threads**

For the submission of the Lab report, please follow the rules published in Portale. For exercises involving C program, it is also required to report the commands used for compilation and for invoking the executable.
**Remember the -pthread option.**

**The deadline for the third laboratory is January 14th, 2021 at 24:00**

**Exercise 1** – Processes vs Threads
Write a C program able to evaluate the maximum number of processes and threads that it is possible to create under the chosen architecture and operating system. Compute the time needed to create and destroy processes and threads, comparing them in a table. Is it better to reboot the computer after such experiments?

**Exercise 2** – Thread lifecycle management
Write a simple program that creates **N** threads, where the value of N is given by command line. Make each thread sleep a random time (from 0 to 10 seconds) and then let the main thread wait for **all N** threads, displaying a message stating which thread ended.

**Exercise 3** – **Shared** counters (mutex)
Write a program that creates **N** threads, where the value of N is given by command line parameter. The program has two shared counters.
Each thread generates a random number and increases one of the two shared counters by 1 according to whether the random number is even or odd.
Report how many mutexes you used and the reason.

**Exercise 4** – Sharing data
Write a program that creates two threads. The main thread will read filename from **stdin** and send the filename to the first thread until it reads '**END**'; the first thread will read the content of file and send the data to the second thread; the second thread must write data into a second file whose name is provided by command line.

**Exercise 5** – Processes using threads
Write a program that creates two children processes for a total of 3 processes. Each child process should now create its own thread. Then, try to send a single byte across threads (from parent process to all the child processes' threads). Are there any limitations in data

exchanging? Eventually, replace one child by an **exec** function call after the creation of the thread; what happens to all threads?

**Exercise 6** – Producers and consumers
Write a program to solve a basic producer-consumer problem. The main threads own a repository of 10 items. Each item is associated to a quantity value, that is 0 when the program starts. A set of N threads (N from command line) simulates the production of the items, whereas other N threads try to sell them. The production is implemented by the following set of steps:

1. For each item, the thread generates a random quantity from 1 to 10, then sleeps 2 seconds before generating the next item quantity.
2. Once all items have gone through point 1, the thread tries to update the repository.

The consuming activity goes following a different approach:

1. The thread picks a **random** item to be served (consumed).
2. For the item, the thread generates a random quantity from 1 to 100, then
3. It checks if the quantity can be served. If it's possible, it serves the item immediately, decreases the quantity of the item and it goes back to point 1. If the quantity is bigger than the one available, the thread must **stop** until a *valid* update to the availability is done. Then it goes back to point 3 to check again.

**Note:** a *valid* update of the availability is an increment of the quantity.
The producer thread does not need to wait for condition variable.
The consumer thread need to wait for a condition variable (item quantity >= number to be served).
For each item, you need, a mutex and a condition variable for consumer.

**Exercise 7** – Math series in a parallel scenario
Write a multi-thread program that evaluates the following math series:

$$\sum_{0}^{N} X^i \big/ Y^i$$

The main program receives the x, y and N values as input (avoid setting a max value for N) and creates the set of necessary threads. Each thread evaluates a single $x^i$ **or** $y^i$ instance, and the main thread compute the final results (division and summation). Once all the threads complete their job, the main program displays the final result. Properly handle issues with data.
Hint: create 2*N threads, pass a pointer into thread as argument and use it to pass result back to main thread.

**Exercise 8** – Prime number search
Write a multi-thread program to search if a number N is a prime number by checking if it is divisible by any number in the range <2, N/2>. The program receives in input the number **N** and the number **P** of threads to use.
The program must create **P** threads and assign to each of them a subset of possible divisors to test. Each thread checks if any of the assigned numbers is a divisor of N.
The main program continuously checks the result of the generated threads.
**Hint: the thread will set a global flag if the number is not prime, all the threads will continue if the flag is not set (not necessary to use a mutex).**

As soon as a thread detects that the number is not prime, the main program must stop the executions of all generated threads and exit.
Once the program it's ready, try its execution with big numbers and different values of P to understand how the execution time changes. Printing the start and end time is a good idea to trace the execution time.

**Exercise 9** – Advanced prime number search
Write now a program to search for all Prime Numbers until a certain maximum value **N**. N comes from command line along with **T** indicating the number of threads to use in parallel. Each thread looks for prime numbers in a specific sub-range of the full search space [1, N] and adds each prime number found into **a shared vector**. Once all have finished, the main thread **sorts** (e.g. bubble sort, merge sort) the vector and print the list of prime number found.
*Hint: Since you do not now the size of the shared vector, resort to an **unsigned int *** to define a pointer to the vector, which will be shared, and then allocate and reallocate the size accordingly using the **malloc/realloc** functions. Remember to **free** the vector once it has been printed, before terminating the program.*

**Skip Exercise 10 for now, wait for further instructions.**
**Exercise 10** – Threads as game (advance data synchronization)
Implement a 5 vs 5 version of the basketball game. Each player is a thread and the referee must be the main thread. Rules are pretty simple:

1. only one thread can own the ball at each time.
2. the thread that owns the ball try to reach the basket, by means of picking a random floating-point number, from 0 to 5. At the same time, only one player of the other team tries to stop him, picking another random floating number, still from 0 to 5. The player with the higher number wins the challenge.
    a. If the player that owns the ball wins, it tries to dunk by picking a floating-point number from 0 to 100.
    If the number below 60, then it misses. In this case, all players pick a random floating-point number from 0 to 1. The bigger one gets the ball and play goes back to point 2.
    If the number is above 60, the player dunks depending on the difference between the random numbers picked by the two players:
        i. if the difference is less or equal to 2, the team gets 2 points.
        ii. If the difference is above 2, the team gets 3 points.
    In both cases, the ball goes to the opposite team, to a random player and play goes back to point 2.
    b. If the player without the ball wins, then it gets the ball and play goes back to point 2.
    c. If they pick the same number, the referee has to call foul:
        i. The referee tosses a coin (randomly generate a 0 or a 1):
            1. If the result is 0, the foul was made by the player with the ball thus the program behavior will be like point b.
            2. If the result is 1, the foul was made by the player without the ball so the player with the ball picks a floating-point number (from 0 to 100) two times, one per tentative. To score single point they have to be greater than 80.

The referee has to end the game after 4 rounds of playing of X seconds elapsed. The X number of seconds per rounds must be provided by the user via command line. The end of each round and of the game have to be asynchronous. When the game ends, write as

much statistics you can gather during the game about: points after each round, total points, number of fouls, etc.
*Hint: Since you will have 11 threads running, with specific roles, you not only need to put mutexes all over the code (still try to reduce their amount not to overcomplicate the code) but also, you'll need conditional wait or similar approaches to signal the "require for an action". Avoid pipes.*

**Exercise 11** – Thread Pool
Write a thread pool handler. The purpose of the handler is to collect has many threads as the user want to run without running them immediately. It releases a new pooled thread to run under a maximum number of running thread constraint (set by the user). The handler should be delivered in the form of a thread that perform all pool operations, e.g., append a thread to the pool, launch a new thread, etc., supported by all data necessary, and by functions that the user should use in place of common pthread_* functions.
Re-implement Exercise 9 using your threadpool library.

// The following part serves only as suggestion of possible implementation.
user producer: produces {func, args}
threadpool handler, consumer: consumes {func, args} -> producer, produces {func, args} -> threads in the threadpool
thread in the threadpool: consumers ->consume {func, args}, execute the func and produce the {result}
threadpool as consumer: consume {result} – gather results to be retrieved by threadpool_wait

```
// You should design the data structure to support following functions.
typdef struct {
// save n_thread ids (for pthread)
// mutexes, condition variables
// counters
} threadpool_t;

int threadpool_init(threadpool_t *p_tp, int n_threads);
```
creates a thread pool containing **n_threads** threads, return -1 if error, return 0 if success.

```
int threadpool_launch(threadpool_t *p_tp, (void*) (*func) (void*), void* args);
```
pick an idle thread in the pool to execute **func** with args, return thread index (inside the pool) if success, -1 if error (this threadpool_launch will block if no idle thread is available right now).

```
void threadpool_wait(threadpool_t *p_tp, int thread_idx);
```
wait for thread index = thread_idx to finish the task.

```
void threadpool_destroy(threadpool_t *p_tp);
```

*Hint: Since we don't want you to re-write the pthread library, you can internally resort to any pthread functions.*

**Exercise 12** – Thread pool with scheduler
Write an advanced version of the previous exercise by adding a simple user level thread scheduler into the pool. The policy for the thread selection from the ready queue must be random, when enabled. The scheduler must support the ready and the running queue.

Once you program the core of the scheduler, submit a working example, using all threads programmed in the previous exercises, e.g. re-implement Ex 9, using threadpool with a random scheduler.

Try to simulate the arrival time of each thread, e.g. insert some delays before task submission (entering ready queue, threadpool_launch) to the pool or before finishes the task (leaving running queue, threadpool_wait), report if anything different can be observed.
At user level, is it possible to emulate/introduce a waiting queue too?