System and Device Programming
Lab on C++ Parallel Programming

Sommario

A. Exam 19 June 2021 – Condition variables (program structure given)

Write a small C++ example where a thread "*admin"* initializes an integer variable *var* to 10 and then waits 3 other threads, which each adds a random number between 0 and 5 to *var*.
The program terminates either when all threads finish or when *var* becomes equal or greater to 15: in both cases the "admin" thread is awakened and prints the final value. Write the code of the function *admin_f()* and *adder_f()* so that the provided main works correctly.
You should use locks and condition variables.

```
std::mutex m;
std::condition_variable adminCV, adderCV;
std::queue<int> taskQueue;
int var = 0;

void admin_f(){
 // COMPLETE
}

void adder_f() {
// COMPLETE
}

int main() {
  int var=0;

  std::vector<std::thread> adders;
  std::thread admin(admin_f);

  for(int i = 0; i < 3; i++){
    srand((unsigned)time(NULL)); //makes seed different for calling rand()
    adders.emplace_back(std::thread( adder_f));
  }

  for(auto& i : adders) {
    i.join();
  }

  adminCV.notify_one();
  admin.join();

  return 0;
}
```

B. Exam 5 July 2021 – Thread communication with future and promises

Write a small C++ example with three threads:
- thread *take_value* takes a number from command line
- thread *check_if_prime* checks whether the number is prime
- thread *give_answer* prints the answer to standard output

Thread communication should be made using promises and futures.

C. Exam 17 January 2022 – Synchronization problem: control of a bar in a parking lot

Write a small C++ program for simulating the control of a bar to let cars enter and exit a parking lot.
For the sake of simulation, consider that a car arrives every *t1* random seconds (1<= t1 <= 3), while a car leaves the parking lot every *t2* random seconds (4<= t2 <=7). If a car cannot enter because the parking lot is full, it will just search for parking somewhere else (hence, no queues will form in front of the bar at the entrance).

The following parameters can be initialized within the program:

- number of places in the parking lot;
- duration of the simulation (in seconds), computed starting from the first car that enters.

The function to put a thread in the sleep status (e.g., for 1 second) is the following one:
```
std::this_thread::sleep_for (std::chrono::seconds(1))
```

Write the code of the program and manage threads synchronization.
Make sure all threads finish running before the main program terminates.

D. Exam 18 October 2021 – Synchronization problem: control of a heating system

Write a small C++ program that simulates the control of a heating system for a room (e.g., an office).
When the program starts, the room temperature is 18.5°, the target temperature is 18°, and heating system is off.

The program must be composed of the following threads:

- a thread "t*argetTemp"* that continuously reads the desired temperature from keyboard (it makes a check of the input every 5 seconds);

- a thread "*currentTemp*" that updates the room temperature every 5 seconds (precision: decimals, e.g., 27.3°, 18.2°, …) according to the following logic:
    - when the heating system is on, it adds 0.3° to the room temperature,
    - when the heating system is off, it subtracts 0.3° to the room temperature.

- a thread "*admin"* manages the heating system with the following logic:
    - it checks the room temperature every 3 seconds,
    - when the heating system is on:
        - if room temperature is more than target temperature+0.2°, it switches the heating system off; otherwise, it does nothing;
    - when the heating system is off:
        - if room temperature is less or equal target temperature+0.2°, it switches the heating system on; otherwise, it does nothing.

When the user inputs -1 (the thread *target_temp* will capture it from the keyboard), the program terminates: make sure that all threads finish running before the main program terminates.

The function to put a thread in the sleep status (e.g., for 1 second) is the following one:

```
std::this_thread::sleep_for (std::chrono::seconds(1))
```

Write the code of the program and manage threads synchronization.

E. Exercise with ASYNC providers and containers

Resorting only to async tasks, solve a matrix multiplication AxB=C.

For matrix multiplication, the number of columns in the first matrix A must be equal to the number of rows in the second matrix B. The result matrix C has the number of rows of A and the number of columns of B (additional information at https://en.wikipedia.org/wiki/Matrix_ multiplication)

In order to test your implementation, wrap it around a program that takes the size of the matrixes A and B as input from keyboard (you might add any check required), randomize the content of both matrixes and saves the result matrix C in an output file.

The following example code shows how to write to a file:

```
#include <fstream>

std :: ofstream outputFile ;

int main() {

outputFile.open ( ” . /output-tasks.txt”);

outputFile << ”Max thread supported ” << std::thread:: hardware
concurrency () << std : : endl ;

//some other code here

outputFile.close() ;

return 0; }
```

Additional requirements and suggestions:

- Use C++ containers only.
- You can use the `std::inner_product` function to compute a sum of products.
- Take advantage of parallelism of asynchronous tasks in doing the computations.
- If you implement the matrixes as vector of vectors, it will be easier doing computations with the transpose of matrix B. Example:
    - `std::vector<std::vector<int>> a, b;`