Project labs: step 4 – A thread-safe shared buffer

Lab target 1: Learn how to develop and use a First-in First-out buffer.

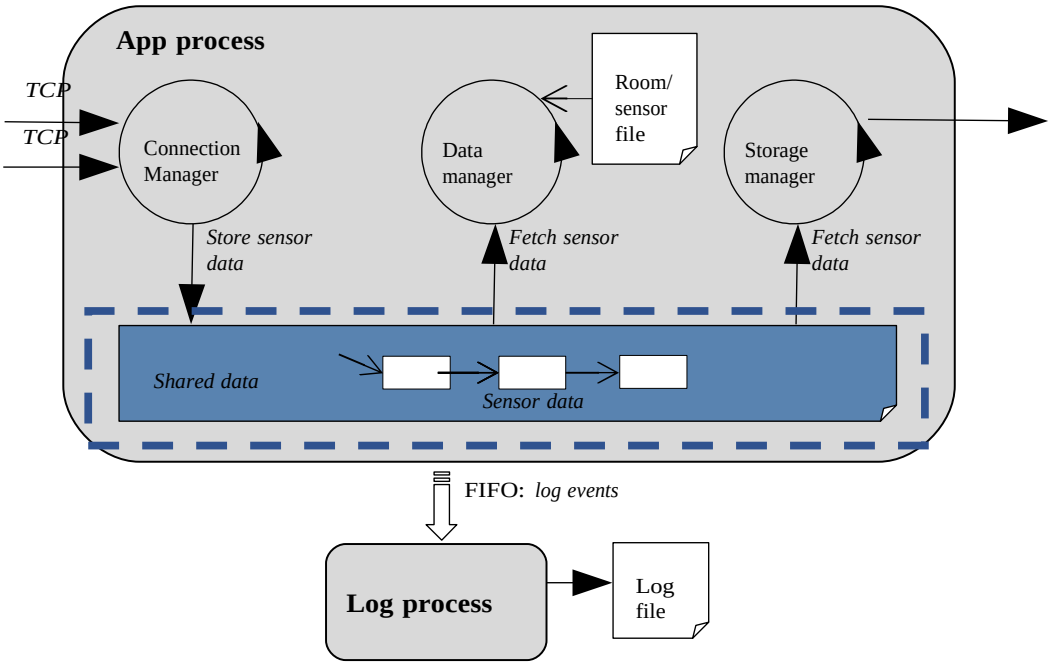
Lab target 2: Apply the concepts of multi-threading to a shared data structure.

Lab target 3: Apply synchronization to develop a thread-safe data structure.

Lab target 4: Submit your solution on Toledo under assignment Milestone 3.

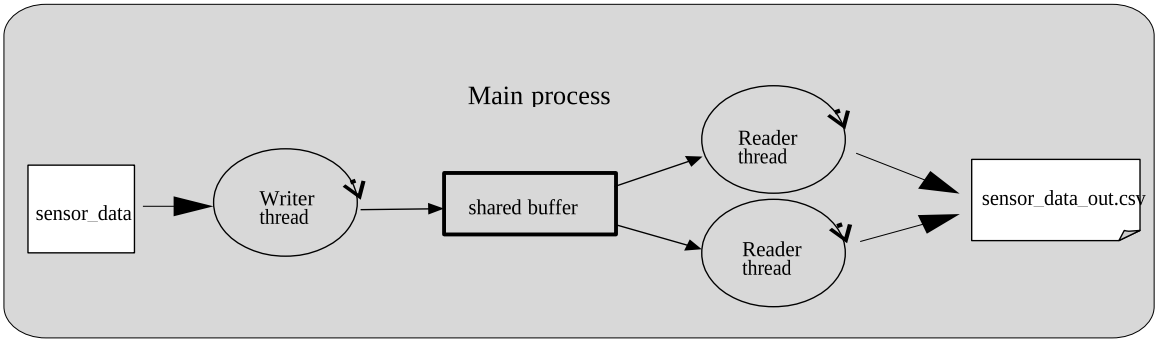
1. Project overview

The relationship of this lab to the final assignment is indicated in blue



HEADS UP: Hand in the solution to this lab as milestone3.zip on Toledo!
- Use the zip build target in the make file to create this zip.
- Include ONLY the code files for this assignment, no previous assignments.
- Include ALL the code files that are needed to build this assignment.
- All code should be in the root folder of the zip file, without sub-folders.
- We build your assignment as follows: gcc *.c -o main.out -lpthread

2. Thread-safe reading from and writing to a shared buffer



Implement a multi-threaded program (main.c) that shares a common buffer with sensor data between 3 threads. Your program creates 2 reader threads and 1 writer thread.

- The writer thread reads sensor data from a single file called 'sensor_data' (see plab1 to create such kind of file and how such file is structured). We also included an example sensor_data file in the startcode zip.
- The writer thread repeats reading a sensor measurement (ID, temperature and timestamp) and inserts this sensor data in the shared buffer. The writer thread inserts 1 sensor measurement every 10 milliseconds. Use the usleep function for this.
- The reader threads repeat removing a complete sensor measurement from the shared buffer and write this data to a common csv file called 'sensor_data_out.csv'. A reader thread waits 25 milliseconds after writing each measurement.
- Reader and writer threads are started in the beginning of the program and run in parallel until all sensor data is processed.
- The shared buffer should be made aware when there are no more measurements by adding an end-of-stream marker to the sbuffer: a dummy sensor reading with sensor id 0 and no other values. The main thread 'waits' on the termination of all reader and writer threads before exiting the process.

The shared buffer can be implemented using several data structures: a circular array, a queue, a dynamic pointer-based data structure, …. For this exercise, a dynamic pointer-based data structure is chosen. The different operators that need to be implemented are defined in '`sbuffer.h`'. In '`sbuffer.c`' you find sample code that implements these operators. However, the implementation in '`sbuffer.c`' doesn't take care of safe 'data sharing' between threads. You have to alter or rewrite the code in '`sbuffer.c`'. Carefully think how you could solve this problem. Which synchronization method is the most appropriate for this situation: one or more mutexes, semaphores, condition variables, or combinations of some of these? Also carefully think about an efficient data locking strategy (locking granularity). Is it always needed to lock the entire data structure for every operation? However, if you are not sure, safety and stability should be chosen over performance and uncertainty.

When *`sbuffer_remove()`* is called and no data is available in the shared buffer, there are two options: the function blocks until data becomes available or the function returns immediately indicating there was no data available (non-blocking). For this exercise, you should have a blocking wait, unless the end-of-stream marker has been detected. Then the function should return there is no data anymore.