System Programming
C programming manual: lab 9
2015 - 2016

Bachelor Electronics/ICT

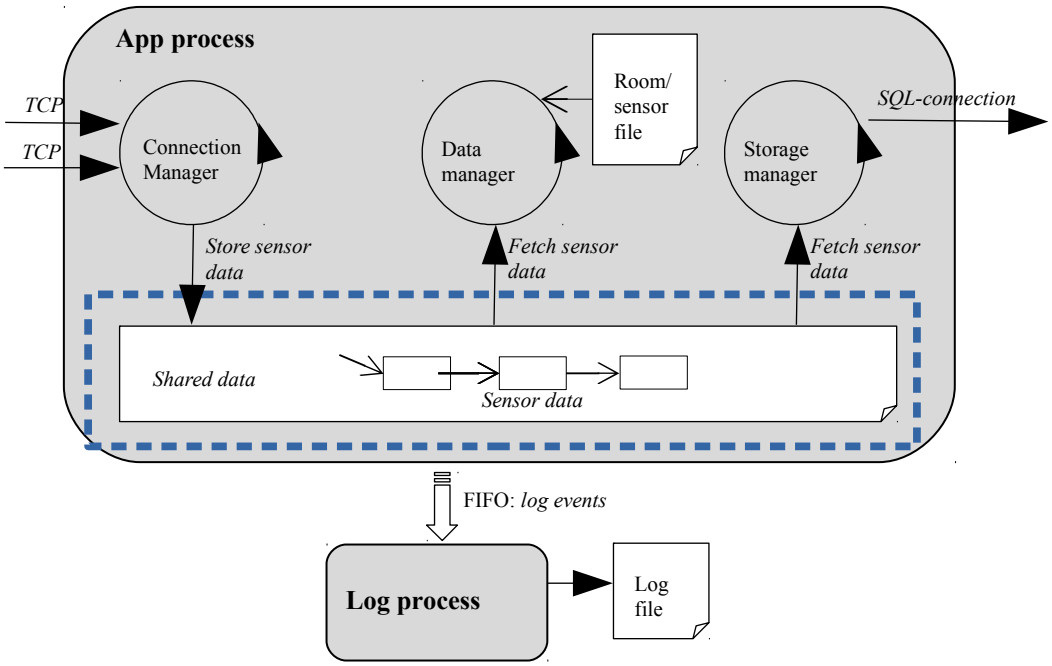
Course coördinator: *Luc Vandeurzen*
Lab coaches: *Jeroen Van Aken*
Stef Desmet
Tim stas
Luc Vandeurzen

Last update: March 25, 2016

C programming

Lab targets:multi-threading programming and thread-safe data structures

For your information
The picture below visually sketches the final assignment of this course. The relationship of this lab to the final assignment is indicated by the dashed blue line.

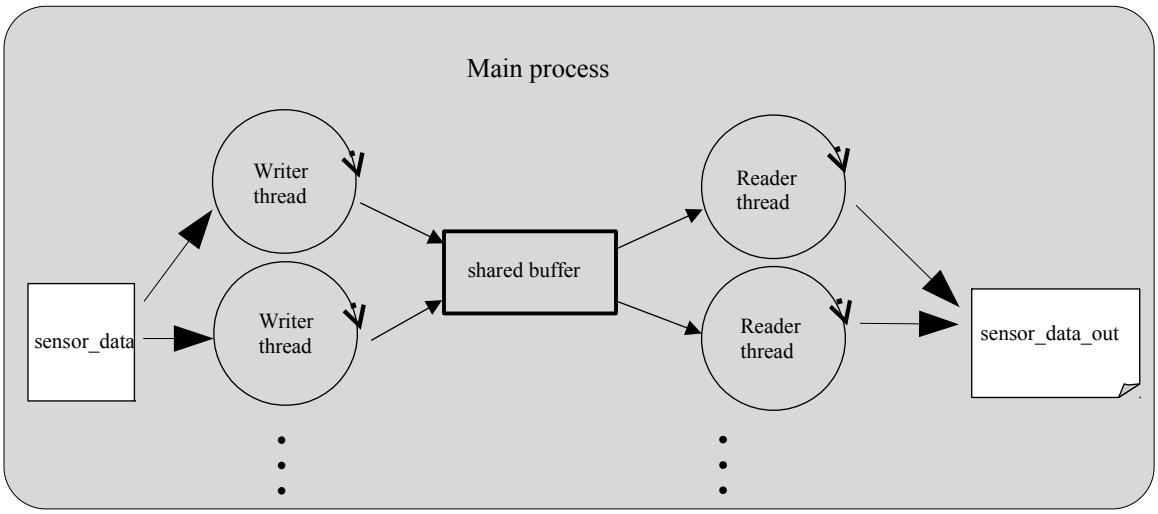


General remark

If you don't have the man pages of pthread-related functions (e.g. pthread_mutex_lock() or pthread_rwlock_destroy()), you should install the packages 'manpages-posix' and 'manpages-posix-dev' on your system.

Exercise 1: *thread-safe reading from and writing to a shared buffer, non-blocking style*

Implement a multi-threading program that share a common buffer with sensor data. The program creates to types of threads: reader threads and writer threads. The writer threads read sensor data from a single file called 'sensor_data' (see previous labs to create such kind of file). A writer thread repeats reading a complete sensor measurement (ID, temperature and timestamp) and inserts this sensor data in a shared buffer. The reader threads repeat reading a complete sensor measurement from the shared buffer and write this data to a common file called 'sensor_data_out'. It should be possible to set the number of reader and writer threads at compile time with the preprocessor directives NUM_READERS and NUM_WRITERS, respectively. Reader and writer threads are started up with a unique ID (e.g. reader 1, 2, 3, etc. and writer 1, 2, 3, etc.) and run in a loop until all sensor data is processed. 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. You are free to (not) use or modify the code in 'sbuffer.c'. However, the implementation in 'sbuffer.c' doesn't take care of the 'data sharing' between threads. Carefully think how you could solve this problem. Which synchronization method is the most appropriate for this situation: one or more mutexes, semaphores, rwlocks, barriers, 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?

Finally, you should notice that 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 functions returns immediately indicating there was no data available (non-blocking). In this exercise, you should implement a non-blocking *sbuffer_remove()*

function. Convince yourself that blocking or non-blocking is less an issue for the *sbuffer_insert()* function.

Remark: what about using standard library I/O functions (fprintf, fscanf, fread, fwrite, …) to read and write to a file shared by multiple threads?

Exercise 2: *thread-safe reading from and writing to a shared buffer, blocking style*
Re-implement the previous exercise but this time use a blocking *sbuffer_remove()* function defined as:

*int sbuffer_remove(sbuffer_t * buffer,sensor_data_t * data, int timeout)*

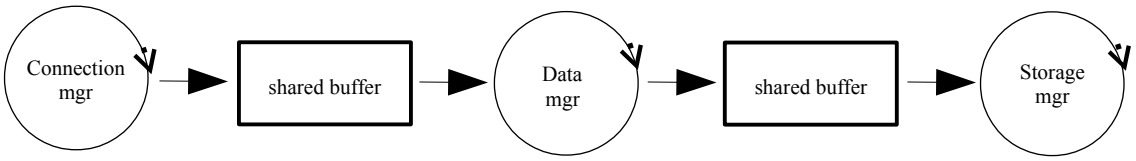
The 'timeout' indicates how many seconds the function may block waiting for data to arrive in the shared buffer. Carefully consider how to implement blocking in an efficient way (a busy wait loop is not considered as efficient).

Remark related to the final assignment

The shared buffer developed in exercise 1 and 2 of this lab, is the code you need for the 'shared data' part of the final assignment of this course. However, there is a *but* …

As you might notice from the picture sketching the final assignment, there are two 'reader threads' in the assignment: a data manager and storage manager thread. Both need to read all sensor data. Hence, if one of them has read sensor data, it can not be 'removed' from the shared buffer because the other manager still needs to process it. Do you see that?

There are several solutions for this problem. A simple solution would be to use a pipeline threading model between both managers, as depicted below. In fact the data manager reads data from a first shared buffer (shared with the connection manager), and after processing the data, writes it to a second shared buffer, leaving it there for the storage manager.



But can you do better than the 'double shared buffer' solution given above?