Exercise 2

Introduction

To protect the sbuffer data structure some synchronization primitives needed to be selected. After analysing the problem and comparing different solutions I decided to work with a read/write lock and a condition variable.

In fact, the first primitive that came to mind when I encountered this problem was the read/write lock. The problem is a typical consumer/producer problem and it is often solved with read/write locks. While the writer, in this case the connmgr, is writing no one can access the sbuffer. Half-written values could be read and the data structure would simply not be atomic. On the other hand when a reader is reading data another reader can perfectly start reading. If I had solved this with mutexes readers would have to wait for each other and the solution would be less efficient.

After I implemented the read/write lock I realised that my readers(sensordb and datamgr) were constantly polling the sbuffer if there wasn't any data. This causes them to be using CPU while the writer needs the time to update the sbuffer. To avoid this I implemented a condition variable. When one of the readers notices that the sbuffer is empty, the condition will be set. While the condition is set the readers are waiting. Since there are 2 readers and a conditional wait can't be implemented with a read/write lock, I used 2 mutexes(1 for each reader) for this. Then when the writer adds new data the condition is cleared and the readers can parse the data in the sbuffer.

Correctness

Now I will evaluate whether my implementation. I will check if deadlocks are avoided, that multiple writers aren't accessing the sbuffer at the same time and if my solution is fair.

To check if deadlocks occur it is good to understand when and how they can occur. A deadlock can happen when multiple threads try to lock the same locks, but in a different order. So for a deadlock to occur there should me multiple locks that are obtained by multiple threads. Since I have 2 mutexes and a read/write lock this is the case. What doesn't happen in my solution is that these threads obtain the mutex/lock in a different order. Both the writer and the readers first try to lock the read/write lock and then (if it's necessary) the mutex(es). This way deadlocks are avoided.

Confirming that multiple writers aren't accessing the sbuffer at the same time is easy because only the connmgr thread accesses the sbuffer to write. A thing that could possibly happen though is that the writer thread starts writing while a reader thread is reading. I prevented this from happening by using the read/write lock.

Finally, I have to analyse whether my solution is fair. In other words if all threads get a fair amount of CPU time. I checked this by inserting print statements in the different threads. When I did this I saw that the prints were intermingled and appeared in equal amounts for each thread. I conclude from this that my solution is fair.

CPU usage

My solution is optimal because the thread using the CPU is always doing something useful. It is not using the CPU to poll data structures, but always parsing or writing data. If there is no data in the sbuffer the reader threads simply wait without using the CPU. The connmgr is polling the TCP socket though, but it works with a timeout and is done by the poll method. This method implements an efficient polling algorithm. The log process is also always polling for log messages on the FIFO. Besides that, by using different threads and processes that can work next to each other the multiple cores of the CPU (if it has multiple cores) can be used. A graphical representation can be seen in figure 1.

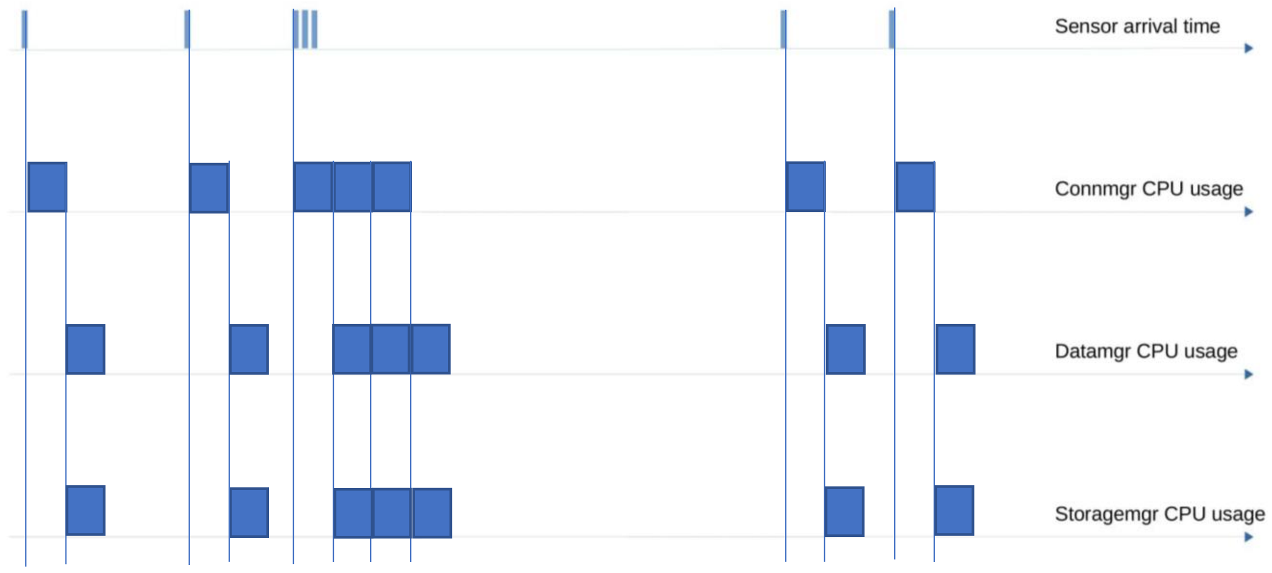

Figure 1 CPU usage of sensor gateway