OS2 Lab 5: Buffer synchronization in C with `pthreads`

imec-DistriNet, KU Leuven Ghent Technology Campus

October 11, 2022

1 Overview

Consider an IoT socket server implemented in C on Linux. The server is set up to receive packets from IoT temperature sensors and has various processing and logging tasks. To facilitate a potentially high packet rate, we parallelize the server's 3 main responsibilities using dedicated threads that communicate via a shared buffer. Figure 1 visualizes this architecture.

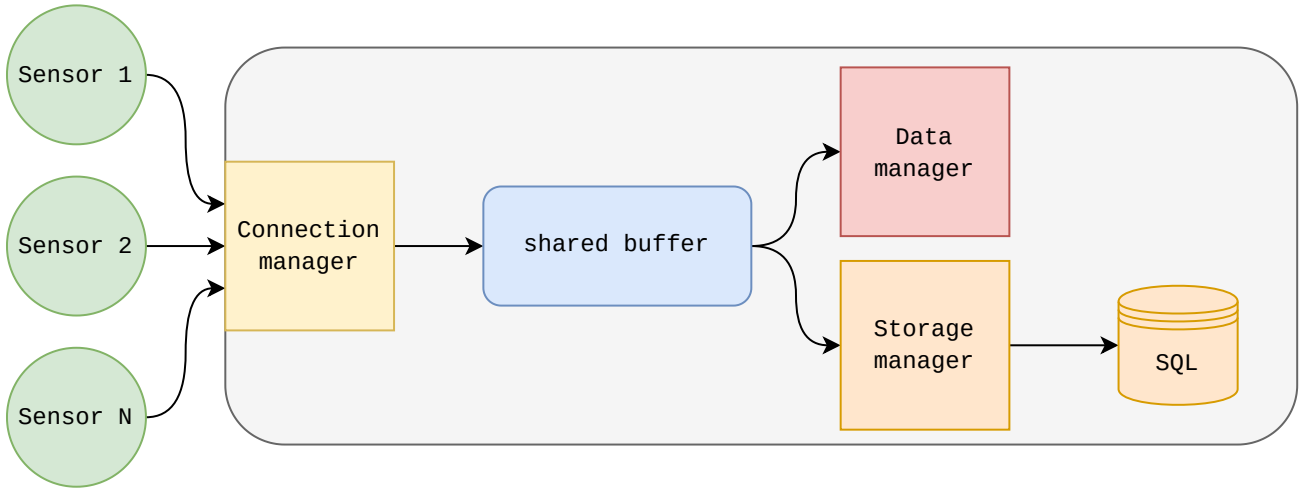

Figure 1: Schematic representation of the server's architecture.

The *connection manager* listens to new packets or connections using Linux's `epoll` interface. Upon receiving a packet, it prepends the incoming data to a singly linked list, the *shared buffer*. Secondly, the *data manager* continuously reads sensor data from the buffer and calculates a running average of the incoming temperature values for each sensor, warning if the temperatures exceed certain limits. Lastly, the *storage manager* also reads sensor data from the buffer, and commits all measurements to an SQL database.

The exact operation of the connection manager, data manager and storage manager are not important. Rather, we will focus on the synchronization of the shared buffer, and how it impacts the performance of the application.

2 Requirements of the assignment

Nearly all code for this lab is already provided, including the shared buffer. However, the current synchronization of the threads on the buffer is done poorly, with a mutex preventing both consumers from processing measurements at the same time. Worse yet, readings are always discarded after being read by a single consumer, preventing the other from processing it. The assignment of this lab is to refactor the code, mostly in `main.c` and `sbuffer.c`, to fix these issues, and satisfy the following requirements:

1. All sensor readings should be seen exactly once by both consumers. Only after both readers have finished processing the packet, it may be removed from the buffer. If the connection manager closes the buffer while there is still data left in it, both readers should still process the remaining data before shutting down the server.

2. None of the threads should suffer starvation. Priority-wise, the data manager should see all incoming readings as fast as possible, to notify an administrator of any dangerous temperature values. On the other hand, the storage manager has no such requirement, as it merely maintains a history of past readings.

3. In the current design, both readers will continuously check for new data ("polling"), causing high CPU usage. Readers should only be active when there is new data for them to read, otherwise sleep.

4. The synchronization of the buffer should be self-contained, contrary to the current implementation. Users of the `sbuffer` should just be able to call `sbuffer` functions without worrying about synchronization, i.e., the `sbuffer_lock` and `sbuffer_unlock` functions have to go.

5. The implementation is currently data race- and memory error-free. Use `valgrind` and/or compiler sanitizers to verify that this remains the case after your changes. Section 3 gives more information on how to do this.

6. (Optional) Although there are two readers right now, it may be worthwhile to experiment with a dynamic amount of readers, while continuing to satisfy all other requirements.

Note that you are free to change any part of the system, including API changes to any of the managers or swapping out the buffer for something more custom, to fit your design better. However, the direct use of the `pthreads` primitives for synchronization remains required. Good places to start are the current critical sections, indicated by calls to the `sbuffer_lock` and `sbuffer_unlock` functions. Notably, look at both entry functions of the data and storage manager threads in `main.c`, which already feature quite a bit of code duplication that should be avoided.

The `pthreads` library offers many fundamental synchronization primitives, which you can freely choose from to finish this assignment. Specifically, `pthread_mutex_t`, `pthread_rwlock_t` and `pthread_cond_t` are good starting points. Besides `pthreads`, use of C11's atomics via `<stdatomic.h>` is also allowed.

3 Building and running

This project is Linux-specific, and has to be built and run on a Linux machine to work. For Windows users, we recommend using Windows Subsystem For Linux (WSL), which you may have already used, to make the assignment in. Follow the instructions at https://code.visualstudio.com/docs/remote/wsl to get started. Alternatively, we provide a Dockerfile from which you can generate a Docker image that provides a similar environment. Remember, however, that your progress will be lost when you kill the container. Either do not kill the container, or set up a bind mount/volume to some local folder on your machine to save your work. Make sure you have access to a Linux shell before continuing to set up this project, come see us if you encounter any issues. Throughout this setup, come get us if you are stuck at any point. Outside the dedicated lab sessions, you can contact us for issues, questions, or suggestions at any time, via e-mail or in-person in our office.

3.1 Obtaining the project sources

In a Linux shell, obtain the project sources by cloning the Git repo at https://github.com/ku-leuven-msec/OS2-shared-buffer-C. Then, we recommend you create a (private) blank repo in your own GitHub account (or somewhere else). Use the following commands to switch your remote tracking `origin` from our remote repo, to yours, and push the initial project. If you are working in a team, only one person has to do this, the others can simply clone from your own remote.

```
# Clone our repo (skip if you already did this)
git clone https://github.com/ku-leuven-msec/OS2-shared-buffer-C.git
# Go into the root of the Git project
cd OS2-shared-buffer-C/
# Show the existing remotes in your local repo (informative)
# This should show our repo as "origin" twice right now, both for "fetch"
    and for "push"
git remote -v
# Rename the current origin (ours) to "upstream"
git remote rename origin upstream
# Add a new origin, i.e., your own
git remote add origin <link-to-your-remote-repo>
# Push the project to your own remote
git push -u origin master
```

3.2 Building the project

Create a directory where you would like the binaries to go, e.g., a `build/` folder inside the source tree, and execute `cmake <path/to/source>` from there. After that, you can use `make` from the build folder to build the project. We show an example below.

```
# In the source root, where main.c is located
mkdir build/
cd build/
cmake ../
make
cd ../
```

3.3 Development environment

Whether you develop natively on Linux, use WSL or Docker, we recommend Visual Studio Code with the `clangd` extension as a development environment. Move the generated `compile_commands.json` file out of your build folder into the root of the source tree (where `main.c` is located) to allow `clangd` to work optimally. If `clangd` shows errors, but your project builds fine (without warnings!), you likely changed some compiler option or added a new file. Update the `compile_commands.json` file in your source root to reflect the changes.

Finally, we recommend using the integrated terminals for a smooth developer experience, including `ctrl +click` file opening etc. Launch a new integrated terminal from the `Terminal -> New Terminal` menu. Continue building and running the project from these integrated terminals during your development.

3.4 Running and testing

Building the project will generate two binaries, that print out usage information when run with no arguments. A typical test looks as follows:

```
# In one terminal, from the directory where main.c is located
build/server 4096 # 4096 is port number of your choice

# In another terminal, same directory
build/sensor 15 1 127.0.0.1 4096
# 15 is the sensor ID (can be any number)
# 1 is the interval (seconds) with which the sensor should send data. If
   you want to stress your buffer, you can set this to 0
# You can start up as many sensors as you like
```

You can launch as many of these sensors as you like. If you want to launch multiple from a single terminal, add a `&` to the end to run the command in the current shell's background. It will still be terminated when you exit the shell. Example:

```
build/sensor 15 1 127.0.0.1 4096 &
```

Finally, because this project is in C, which is memory-unsafe, we recommend checking your project for memory errors regularly. Uncomment the `-fsanitize=address` lines in the top-level `CMakeLists.txt` file to use your compiler's address sanitizer mode. Alternatively, you can use other sanitizers too, notably `-fsanitize=thread` and `-fsanitize=undefined`, to check for race conditions and undefined behavior respectively. Come see us if you encounter any issues.