Technical University of Berlin Specialty Complex and distributed IT systems Summer semester 2024	**Practical task 2** to - System programming - Prof. Dr. Odej Kao
Submission deadline:	**12.07.2024 23:59 h**

In this homework assignment you are to implement a thread-safe ring buffer for messages of variable size.

Task 1: Threadless ring buffer (12P)

You are to create a ring buffer for messages of variable size, which is initially implemented without thread safety. This allows you to test and ensure the basic logic and functionality of the ring buffer. This means in detail: You can initially assume that no two processes are accessing the ring buffer at the same time (neither reading nor writing).

Use the predefined data structures and functions from the file *include/ring- buf.h* and add the code to the marked positions in the file *src/ringbuf.c*.

How the ring buffer works

The information about the ring buffer is stored in a variable of the type *rbctx t.* This struct contains the following elements:

- `uint8_t* begin` Start of the ring buffer memory
- `uint8_t* end` address one step after the last usable byte in the ring buffer
- `uint8_t* read` Position of the read pointer
- `uint8_t* write` Position of the write pointer
- `pthread_mutex t mutex read` Mutex variable to be used by the reader threads to avoid simultaneous reading
- `pthread_mutex t mutex write` Mutex variable to be used by the writer threads to avoid simultaneous writing
- `pthread_cond t signal read` Signal for which reader threads are waiting
- `pthread_cond t signal write` Signal for which writer threads are waiting

Initially, both read and write pointers are located at the very beginning of the ring buffer (see Figure 1).

If data is to be written to the ring buffer, it is written to the position of the write pointer. The write pointer is then moved so that it is one position after the written data. The user data is now positioned between the read pointer and the write pointer. Only this data can be read out again, see Figure 2.

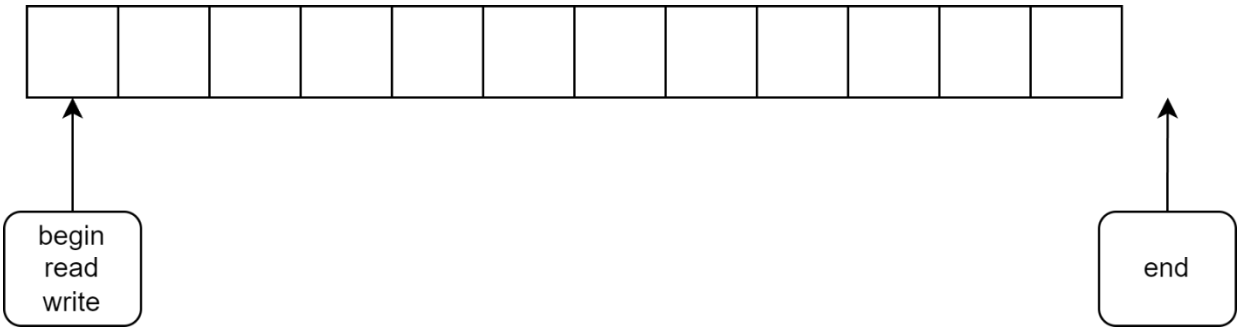

Figure 1: Initial situation of a buffer of length 12 bytes

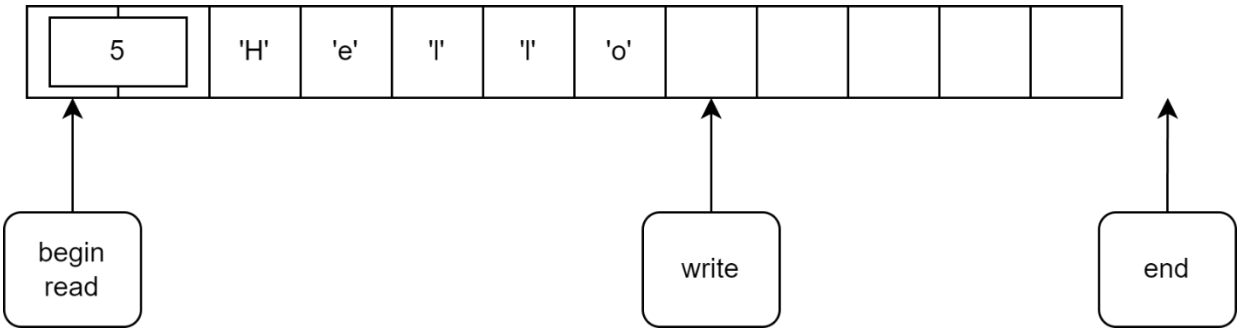

Figure 2: Ringbuffer after calling `ringbuffer write` with the `message` „Hello" and a `message_length` of 5. It **is assumed here that 2 bytes are used to store the lengths. will be. That will be more in the real program!**

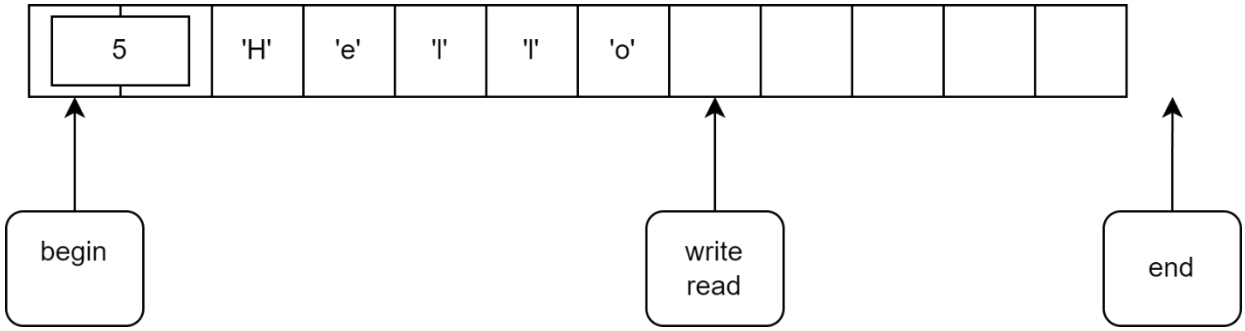

Figure 3: Ring buffer after calling `ringbuffer read`. The `buffer len ptr` parameter must be at least 5.

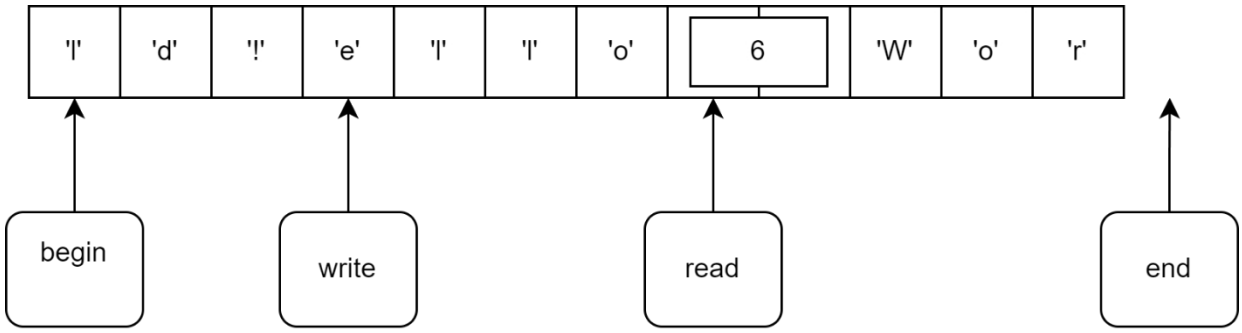

Figure 4: Ringbuffer after calling `ringbuffer write` with the `message` „World!" and a `message_len` from 6

When data is read out, the read process starts at the read pointer and may only go as far as the write pointer. After the read process, the read pointer is set to one position after the last byte read, see Figure 3.

It can happen that the space between the write pointer and the end of the buffer is smaller than the message to be written. In this case, the write process should take place up to the end of the buffer and be continued at the beginning of the buffer (if this is permitted, i.e. no overwriting would occur), see Figure 4. The read process must also be continued at the beginning if messages are to be read that were written to the buffer as described above.

In this way, the read and write pointers move further and further around the ring. It may al-
However, it must never happen that the write pointer „rounds" the read pointer, as otherwise data that has not yet been read will be overwritten. Similarly, the
Read pointer must not „overtake" the write pointer, as otherwise data that has already been read will be read.

Data format

The messages should be stored in the ring buffer in the following format: The first *size t* bytes serve as the header or prefix of the actual message. You save the length of the message in these first bytes. The actual message is stored after this prefix. This structure should make it easier to store messages of variable length.

Recognize and process sounds when reading.

Functions to be implemented

Please also refer to the *lib/ringbuf.h* file for information on the functions:

- `ringbuffer_init` This function is intended to create the initial situation as shown in Figure 1. To do this, it is passed a pointer to a ring buffer context structure in which the relevant data is to be saved. The memory position of the ring buffer is also passed to the function. The memory for this buffer must be reserved beforehand.
- `ringbuffer_write` This function executes the write steps as described above.
- `ringbuffer_read` This function is passed a buffer into which the message to be read is to be written. Only one message at a time should be read. No messages in the ring buffer may be skipped. The parameter `buffer len ptr` specifies how long the passed buffer is and therefore also the maximum length of the message to be read. At the end of the read process, the actual length of the read message should be written to the memory address to which this variable points.
- `ringbuffer_destroy` Destroys data structures that were created during initialization.

Before you start editing, it is advisable to consider the possible cases that can occur when messages are written to and read from the ring buffer. Specifically, these are the cases:

- **read_pointer > write pointer**: The read pointer is located behind the write pointer.
- **read_pointer == write pointer**: The read pointer is the same as the write pointer.
- **read_pointer < write pointer**: The read pointer is located before the write pointer.

Take these cases into account in your implementation. To make testing easier, we have
We have already provided you with test programs (in the *test unthreaded no wrap* and *test unthreaded wrap* folders).

Task 2: Threadsafer ring buffer (12P)

In this task, expand your ring buffer so that several processes can access the ring buffer at the same time. Consider which critical sections exist in your ring buffer and how you can protect them.

For implementation, it is recommended that you first protect the functions with *lock* and *unlock*. Later you should add *signal* and *wait* to your solution in order to avoid busy waiting.

To prevent a write or read operation from blocking endlessly, familiarize yourself with the function *pthread cond timedwait*. This function enables a thread,

to wake up after a certain time, even if no signal has been sent. If you wake up due to a timeout (select one second here), you can exit the function by returning the corresponding return value.

To make testing easier, we have already provided a test program (in the *test_threaded* folder).

Task 3: Simple network daemon (6P)

Especially in the area of data stream processing, it is often necessary to use efficient buffers (such as the thread-safe ring buffer implemented by you). In this task, you are to implement a simple network daemon. In its basic structure, a network daemon processes incoming messages from various sources, such as network packets from port 80 (HTTP) or 443 (HTTPS). To do this, it stores the packets in a buffer, which are then read from the ring buffer by one or more (worker) threads and processed further. In the classic case, the processing involves port forwarding, logging, filtering, firewalling, etc. However, this task is less about the network aspect and more about the interaction between producers and consumers (as you learned in the lecture).

Use the predefined data structures and functions from the *daemon.c* file.

We have already provided you with the functionality that writes to the buffer. Your task now is to implement the corresponding processing threads (readers). In concrete terms, this means that you should take care of correct forwarding and simple firewalling.

Data format of the packets

The messages stored in the buffer are structured as follows: The first *size_t* bytes of the message indicate the port from which the message originates. The following *size_t bytes indicate* the port to which the message is to be forwarded. The following *size_t bytes indicate* which packet of the original file is involved. This is required to correctly reassemble the message on the consumer side. The actual message follows after this metadata.

Firewalling/Filtering

You should now filter those messages for which

- the source port is the same as the destination port.
- are the source or destination port 42.
- is the sum of the source and destination port 42.
- The actual message contains the string *malicious*. There can be any number of other characters between the letters of the string. However, the order of the letters must be observed. Upper and lower case should be observed (i.e. *malicious* but not *malicious* is recognized).

Forwarding

In this task, it is not necessary to deal with network communication (ports, sockets, etc.). Instead, we simulate network traffic via files. The writing threads write data from files to the ring buffer, sending the files in smaller packets with random time intervals. This simulates incoming network traffic on different ports. You do not have to forward the processed messages to the actual target ports. Instead, you should attach the messages to an output file with the name *zielportnummer.txt*. A message directed to port 2 should be attached to the file *2.txt* (without newline). They only attach the actual content (not the metadata) to the file. Messages that are filtered, i.e. are not to be forwarded, are simply discarded.

To simplify testing, we have already provided a test program (in the *test daemon* folder).

Important considerations

- Remember to ensure thread safety when writing to the output files. Two threads writing to the same file at the same time can lead to unpredictable results.
- Consider how to handle this if several processing threads process packets from the same source port. In this case, the packets must be written to the output file in the correct order. To do this, use the *lock*, *while*, *wait*, *work*, *unlock* pattern, and make yourself known with the function *pthread_cond broadcast* familiar.
- The race condition that occurs when two different sources write to a target at the same time can be ignored. Concrete example: If port 1 sends two packets (*ab*, *cd*) and port 2 sends two packets (*ef*, *gh*) to port 3, the order of the written packets does not matter. Possible valid outputs are: *abcdefgh*, *abefcdgh*, *abefghcd*, *efghabcd*, *efabghcd* and *efabcdgh*. However, *aefbcdgh, for* example, is invalid because the packet *ab* is interrupted, as is *cdabefgh* because the sequence of *ab* and *cd* is not correct. It is important that the order of the packets (within a source) remains correct and that the segments are not interrupted. However, the sequence between the packets of different sources can be arbitrary.
- **Attention:** The daemon is already implemented in such a way that it is automatically terminated after a certain time (in this case 5 seconds). An attempt is made to terminate all threads properly by joining all threads with *pthread join*. However, since your processing threads (should) run in an infinite loop, this line is never reached. To terminate your threads from the *outside* (i.e. from the main thread), we use the *pthread cancel* function. Familiarize yourself with this function and all associated functions such as *pthread setcancelstate*, *pthread setcanceltype* and *pthread testcancel.* In your submission, the processing threads must be terminable with *pthread cancel.* During development, you can also terminate the test program (and the associated processing threads) manually with *CTRL+C*, as it is quite possible that everything is already working correctly, but the processing threads cannot yet be terminated *externally.*

Notes:

- **Descriptions of the functions:** Further descriptions of the individual functions can also be found in the .h files in the include folder.
- **Specifications:** Please do not change existing data structures, function names, header files, ... not. Failure to do so may result in points being deducted. You are welcome to define additional auxiliary functions or data structures to make implementation easier for you.
- **Makefile:** Please use the Makefile from the default for this task. To do this, run `make` in the main directory. *make* compiles the project with *clang* under the *build* folder. The compiled files can be executed from a terminal in the shell. To execute a program, use the following command: *./pathto/executable*. Example for the simple unthreaded test:
  `./build/test_unthreaded_wrap/test_simple`.
  For tests where files must be passed as arguments, use :
  ./pathto/executable pathto/file1 pathto/file2. Under Ubuntu, *make* and *clang* can be installed with the command `sudo apt install make clang`. The `make clean` command deletes compiled files. You are welcome to add further flags to the master file or adapt it in other ways. However, your program should remain compilable with the default Makefile. Again, please refer to the README.
- **Memory leaks**: Finally, check your program for memory leaks to evaluate whether the entire memory allocated by the scheduler has been released again. The valgrind command line tool is recommended here[1] or leaks[2] under MacOS. Memory leaks lead to **points** being **deducted**! Remember that you have to use valgrind on the execution of *./pathto/executable*, **not** on *make* (you want to find the memory leaks in your program, not in *make*.

Delivery:

- **Pack** the *src* folder with the edited **.c* files (as in the default) into a zip archive with the name *submission.zip*. Header files (**.h*) should not be changed and therefore should not be submitted. You can use the make target *make pack* for this. The *zip* tool must be installed on your system for *make pack* to work! Then upload the archive to ISIS.
- **Important:** It is not necessary to personalize the archive and its contents with your name or matriculation number. The submission is automatically assigned to your ISIS ac- count!

[1] http://valgrind.org/
[2] https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingLeaks.html