Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2024	**Praxisaufgabe 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin: **12.07.2024 23:59 Uhr**	

In this assignment, you are tasked with implementing a thread-safe ring buffer for messages of variable size.

Aufgabe 1: Threadloser Ringbuffer (12P)

In this assignment, you are tasked with implementing a thread-safe ring buffer for messages of variable size. Initially, you will implement it without thread safety to ensure basic logic and functionality of the ring buffer. Specifically, you can assume that no two processes will access the ring buffer simultaneously (neither reading nor writing).

You will use predefined data structures and functions from the file `include/ringbuf.h` and add code in the marked sections of the file `src/ringbuf.c`.

Functionality of the Ring Buffer:

Information about the ring buffer is stored in a variable of type `rbctx_t`. This struct includes the following elements:

- `uint8_t *begin`: Start of the ring buffer memory.
- `uint8_t *end`: Address one step past 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 used by reader threads to prevent simultaneous reading.
- `pthread_mutex_t mutex_write`: Mutex variable used by writer threads to prevent simultaneous writing.
- `pthread_cond_t signal_read`: Signal that reader threads wait on.
- `pthread_cond_t signal_write`: Signal that writer threads wait on.

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

When data needs to be written into the ring buffer, it is written to the position indicated by the write pointer. After writing, the write pointer is moved forward to the next position after the written data. This leaves valid data between the read pointer and the write pointer, which are the only data that should be read (see Figure 2).

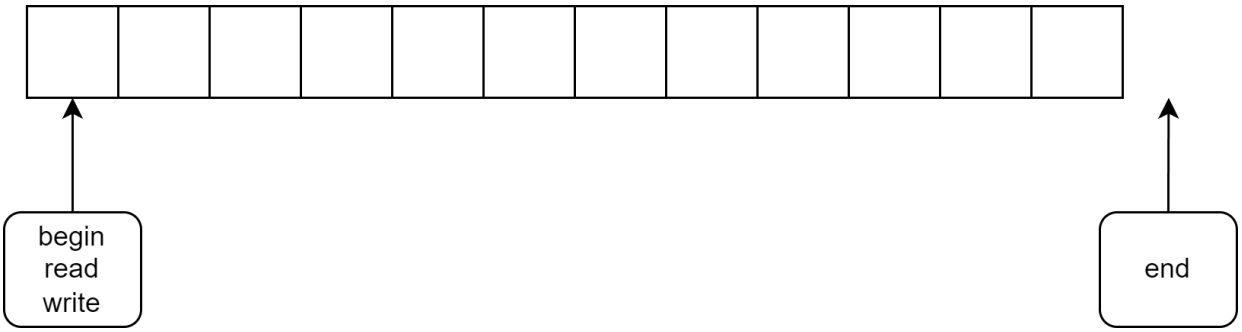

Figure 1: Initial state of a buffer of length 12 bytes

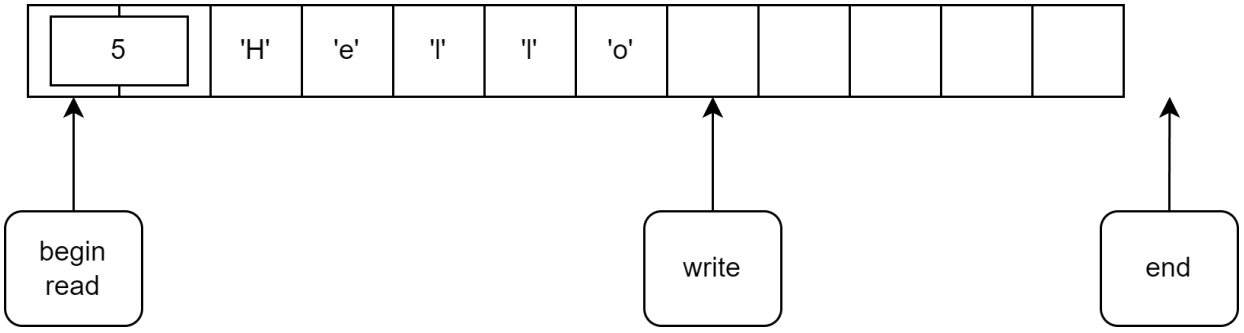

Figure 2: Ring buffer after calling ringbuffer write with the message 'Hello' and a message length of 5. Here, it is assumed that 2 bytes are used for storing the length. This will be more in the actual program!

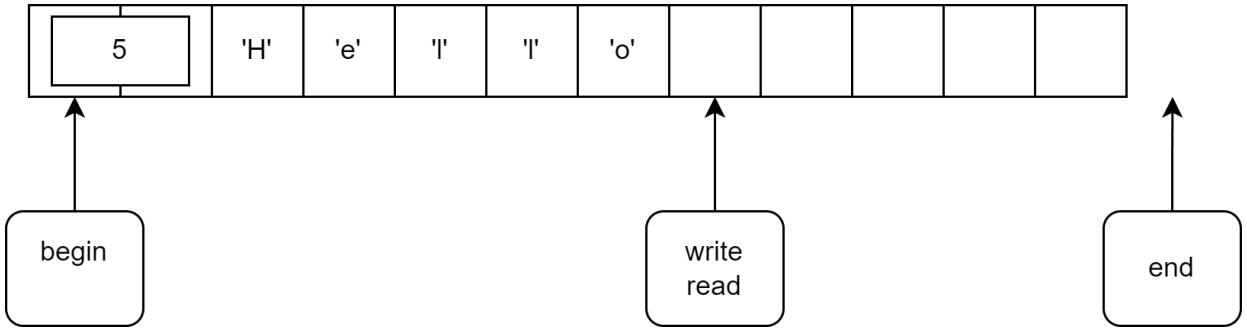

Figure 3: Ring buffer after calling ringbuffer read. The parameter buffer len ptr must be at least 5.

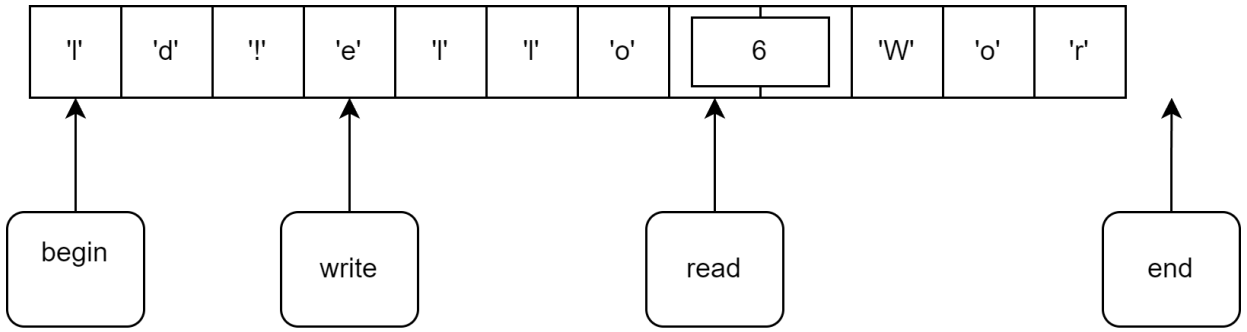

Figure 4: Ring buffer after calling ringbuffer write with the message 'World!' and a message length of 6.

When data is read from the buffer, the reading process starts at the read pointer and can proceed up to the write pointer at most. After the read operation, the read pointer is advanced to a position just past the last byte that was read, as shown in Figure 3.

It can happen that the space between the write pointer and the end of the buffer is smaller than the size of the message to be written. In this case, the writing process should wrap around to the beginning of the buffer if allowed (i.e., it won't overwrite unread data), as illustrated in Figure 4. Similarly, the reading process must also wrap around to the beginning when reading messages that were written in this manner.

This way, the read and write pointers continuously move around the ring. However, it's crucial to ensure that the write pointer never overtakes the read pointer, which would overwrite data that hasn't been read yet. Similarly, the read pointer should not surpass the write pointer, as this would result in reading data that has already been read.

This management ensures that the ring buffer operates correctly and efficiently, handling both full and empty conditions while preventing data loss or corruption.

The data format

Messages in the ring buffer will be stored in the following format: The first `size_t` bytes serve as the header or prefix of the actual message. Within these initial bytes, the length of the message is stored. Following this prefix, the actual message content is stored. This

structure allows for the recognition and processing of messages of variable lengths during reading.

Functions to Implement

Please refer to the file `lib/ringbuf.h` for additional information on the functions. Here are further details about the functions:

- `ringbuffer_init` This function should establish the initial state as shown in Figure 1. It is given a pointer to a ring buffer context structure where the relevant data will be stored. Additionally, the memory location of the ring buffer is passed to the function. The memory for this buffer must already be allocated beforehand
  - `ringbuffer_write` This function performs the write steps as described above.
  - `ringbuffer_read` This function is given a buffer where the message to be read should be written. Only one message should be read at a time. No messages in the ring buffer should be skipped. The parameter `buffer_len_ptr` indicates the length of the provided buffer and thus the maximum length of the message to be read. The memory address pointed to by this variable should store the actual length of the message read at the end of the read operation.
  - `ringbuffer_destroy` Destroys data structures that were created during initialization.

Before you begin implementation, it's advisable to consider the possible scenarios that can occur when writing messages to and reading from the ring buffer. Specifically, these scenarios are:

- **read_pointer > write_pointer**: The read pointer is ahead of the write pointer.
  - **read_pointer == write_pointer**: The read pointer is equal to the write pointer.
  - **read_pointer < write_pointer**: The read pointer is behind the write pointer.

Take these scenarios into account when implementing your solution. To facilitate testing, test programs have already been provided to you in the directories `test_unthreaded_no_wrap` and `test_unthreaded_wrap`.

Task 2: Thread-Safe Ring Buffer (12P)

Expand your ring buffer in this task so that multiple processes can access the ring buffer simultaneously. Consider which critical sections exist in your ring buffer and how you can protect them.

To implement this, it is advisable to first protect the functions with lock and unlock. Later, you should extend your solution with signal and wait to avoid busy-waiting.

To ensure that a write or read operation does not block indefinitely, familiarize yourself with the function `pthread_cond_timedwait`. This function allows a thread to wake up after a certain period, even if no signal was sent. If you wake up due to a timeout

(choose one second for this), you can exit the function, returning the corresponding return value.

To facilitate testing, we have already provided a test program (in the folder `test_threaded`).

Aufgabe 3: Simpler Netzwerkdaemon (6P)

In the field of data stream processing, it is often necessary to use efficient buffers (such as the thread-safe ring buffer you have implemented). In this task, you will implement a simple network daemon. The basic structure of a network daemon involves processing incoming messages from various sources, such as network packets from port 80 (HTTP) or 443 (HTTPS). These packets are stored in a buffer and then read and processed by one or more (worker) threads from the ring buffer. Typically, this processing involves port forwarding, logging, filtering, firewalling, etc. However, in this task, the focus is less on the networking aspect and more on the interaction between producers and consumers (as you learned in the lecture).

Use the provided data structures and functions from the file `daemon.c`.

We have already provided the functionality for writing into the buffer. Your task is to implement the corresponding processing threads (readers). Specifically, this means ensuring the correct forwarding and implementing basic firewalling.

Datenformat der Packets

The messages stored in the buffer are structured as follows:

- In the first `size_t` bytes of the message, the source port from which the message originated is stored.
- In the next `size_t` bytes, the destination port to which the message should be forwarded is stored.
- In the following `size_t` bytes, the packet ID from the original file is stored. This is necessary to correctly reassemble the message on the consumer side.
- The actual message content follows these metadata fields.

Firewalling/Filtering

You need to filter out messages based on the following criteria:

- The source port is the same as the destination port.
- The source or destination port is 42..
- The sum of the source and destination ports is 42.
- The actual message contains the string "malicious". Any number of other characters can appear between the letters of the string, but the order of the letters must be maintained. Case sensitivity must be observed (i.e., "malicious" but not "Malicious" is detected).

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 into the ring buffer, splitting the files into smaller packets sent at random intervals. This simulates incoming network traffic on different ports. The processed messages do not need to be forwarded to the actual destination ports. Instead, you should append the messages to an output file named `destination_port_number.txt`. For example, a message directed to port 2 should be appended to the file `2.txt` (without newline). Only the actual content (not the metadata) should be appended to the file. Messages that are filtered out, i.e., not forwarded, should simply be discarded.

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

Important Considerations

- **Thread Safety**: Ensure thread safety when writing to output files. Two threads writing to the same file simultaneously can lead to unpredictable results.
- **Order of Packets**: When multiple processing threads handle packets from the same source port, the packets must be written to the output file in the correct order. Use the lock, while, wait, work, unlock pattern and familiarize yourself with the function `pthread_cond_broadcast`.
- **Race Condition Handling**: You can ignore the race condition that occurs when two different sources write to the same destination simultaneously. For 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. Valid outputs include: `abcdefgh`, `abefcdgh`, `abefghcd`, `efghabcd`, `efabghcd`, and `efabcdgh`. Invalid outputs include: `aefbcdgh` (since the packet `ab` is interrupted) and `cdabefgh` (since the order of `ab` and `cd` is incorrect). The key is to maintain the correct order of packets within a source and not interrupt segments. The order between packets from different sources can be arbitrary.
- **Daemon Termination**: The daemon is already implemented to automatically terminate after a certain period (in this case, 5 seconds). It attempts to properly join all threads using `pthread_join`. However, since your processing threads run in an infinite loop, this line is never reached. To terminate your threads from the outside (i.e., from the main thread), use the function `pthread_cancel`. Familiarize yourself with this function and related ones like `pthread_setcancelstate`, `pthread_setcanceltype`, and `pthread_testcancel`. In your submission, processing threads must be terminable using `pthread_cancel`. During development, you can manually terminate the test program (and the associated processing threads) with CTRL+C, as it is possible that everything is working correctly except that the processing threads cannot be terminated from the outside.

Notes:

- **Function Descriptions**: Detailed descriptions of individual functions can be found in the .h files in the `include` directory.
- **Guidelines**: Please do not change existing data structures, function names, header files, etc. Ignoring this may result in point deductions. You are welcome to define additional helper functions or data structures to facilitate implementation.
- **Makefile:** Use the provided Makefile for this task. Run `make` in the main directory. `make` compiles the project with `clang` under the `build` directory. The compiled files can be executed from a terminal shell. To run a program, use the following command: `./pathto/executable`. Example for the simple unthreaded test: `./build/test unthreaded wrap/test simple`.
  For tests where files need to be passed as arguments, use: *./pathto/executable pathto/file1 pathto/file2*. On Ubuntu, `make` and `clang` can be installed with the command `sudo apt install make clang`. The command `make clean` deletes compiled files. You can extend the Makefile with additional flags or make other adjustments. However, your program should still be compilable with the provided Makefile. Also, refer to the README for further details.
  - **Memoryleaks:** Finally, check your program for memory leaks to ensure that all allocated memory is freed. The command-line tool `valgrind` (or `leaks` on macOS) is recommended for this. Memory leaks will result in point deductions! Remember to apply `valgrind` to the execution of `./pathto/executable`, not to `make` (you want to find memory leaks in your program, not in make).

Submission

Package the `src` folder with the modified *.c files (as specified) into a zip archive named* `submission.zip`. *Header files (*.h) should not be modified and therefore not included in the submission. You can use the make target `make pack` for this. For `make pack` to work, the `zip` tool must be installed on your system. Upload the archive to ISIS.
Important: It is not necessary to personalize the archive and its content with your name or matriculation number. The submission will be automatically assigned to your ISIS account.

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