Chatroom using a character device

Problem: Enable two or more processes to chat with each other using a character device (say /dev/chatroom).

Details

1. Implement a kernel module and a user-level program.
2. Your kernel module should create a character device called /dev/chatroom when inserted into the kernel and implement the open(), close(), read() and write() functions.
3. A user-level process can join the chatroom using the open() system call on /dev/chatroom as soon as the process is started.
4. To send a chat message to other processes in the chatroom, a process uses the write() system call on /dev/chatroom. The message is sent to all the other processes which have joined the chatroom.
5. To receive a message, a process uses the read() system call on /dev/chatroom.
6. A process can exit the chatroom by typing “Bye!”, which invokes the close() system call on /dev/chatroom.

Here’s an example of a chat timeline between three processes.

Time	Process 1	Process 2	Process 3
1	$ joinchat P1 You joined as P1		
2	P2 joined the room	$ joinchat P2 You joined as P2	
3	P1> Hello P2!	P1: Hello P2!	
4	P2: Hello back to you P1!	P2> Hello back to you P1!	
5	P3 joined the room	P3 joined the room	$ joinchat P3 You joined as P3.
6	P3: Anyone here?	P3: Anyone here?	P3> Anyone here?
7	P1> Hello P3, I am here! P2: Hey P3, P2 here!	P2> Hey P3, P2 here! P1: Hello P3, I am here!	P2: Hey P3, P2 here! P1: Hello P3, I am here!
8	P1> Bye!	P1 left	P1 left

Please pay attention to the following points:

- In Step 7 above, messages from different processes don’t get mixed up in P3’s stdout.

    - This means that you will implement message-oriented communication using a character device (which is technically for byte-stream communication). Such mixing of message types is not ideal, but OK for this assignment.
    - You will need to maintain several FIFO (first-in-first-out) queues in the kernel module - one FIFO queue for each process that has joined the chatroom.
    - Each FIFO queue stores messages that have not yet been read by the corresponding user-level process.
    - An easy way to implement this data structure is to use the kfifo data structure, which is provided by the Linux kernel API. It's fairly easy to learn.
        - Here is a kfifo example (adapted from examples by the original author of the new kfifo API).
        - Here's a helpful article from 2009 that describes these kfifo interfaces when they were proposed initially. It's a bit different now.
    - Create the FIFO queue for each process in the open() function when it joins /dev/chatroom. Correspondingly, cleanup (delete/free) the FIFO queue in the close() function when a process leaves the chatroom.
- You may notice in the above example that messages appear to be delivered "instantly" to the receiver processes on their standard output (stdout), even though the receiving process is itself waiting for keyboard input on its own standard input (stdin). The way to implement this is to create two threads in each user-level process:
    - one thread for reading from stdin and writing messages to /dev/chatroom and
    - another thread for reading messages from /dev/chatroom and printing to stdout.
- The order of message arrival from different processes are non-deterministic. For example, in Step 7 above, P3 receives P2's message before P1's message, but the messages could also have arrived the other way around. Different receivers may receive concurrent messages in different order and that is OK.
- You don't need to maintain a chat history for late arrivals. E.g. P3 does not see older messages that were exchanged before Step 5 above.
- A minor detail: how to send the processes names (P1, P2, P3) to the kernel module.
    - You can replace P1, P2, P3 etc by the process ID of each process.
    - Process ID can be retrieved in the kernel using variable get_current()->tgid. The function get_current() returns a per-CPU pointer to task_struct of the process that is currently running on the CPU and tgid is its process ID
    - A better way is to implement an ioctl() interface for your character device to initialize the process' name, and even implement "join" and "leave" commands, if you want. Implementing ioctl() is optional for this assignment, but I encourage you to learn it, as it's a very powerful interface for device drivers.
- Hint: Start by implementing and testing a 2-process chat first, so that you have something working to submit, and then extend it to a multi-process chat.

Useful links

- Assignment 3 lecture video (April 11 2022)
- Slides from class lecture
- kfifo example shown in class
- kfifo API