Multi-Process Multi-Threaded Chat Server

Setting up Inter-Process Communication:

Before the child processes are created, shared memory is created and attached by the parent process. The parent process also creates an array of socket pairs to facilitate Inter Client communication. Semaphores are created and initialized to synchronize the actions of multiple threads on shared memory and socket pairs.

Creation of Processes:

After the IPC has been successfully set up, the parent process spawns N children and stores their process id in a global array. **Each process is given a unique p_id ranging from 0 to N-1**. After the child processes have been created, the parent's only job is to wait for signals.

On receiving SIGCHLD signal:

The signal handler for SIGCHLD loops through the global process id array and calls waitpid() on each process id in a nonblocking manner. If the waitpid() call is successful, a new process is spawned, and its process id is stored in the array at the index where waitpid() was successful.

On receiving SIGINT signal:

The SIGCHLD handler is removed. The parent sends SIGKILL to all the processes in the process id global array. All IPC resources (semaphores and shared memory) are released. The process exits.

Execution of Child Processes:

Each Child process keeps track of 1. How many clients it has finished handling. 2. How many more connections can it accept. These two pieces of information are maintained in variables, access to which by threads is synchronized with the help of a mutex.

Before spawning threads, the Child process creates a socket pair array of length T; the use of this array is described later.

The child spawns T threads and then enters an infinite reading loop on a socket from the socket pair array created by the parent process.

Execution of Threads:

Each thread is given a unique(within its own process) t_id ranging between 0 and T-1. Each thread runs an infinite loop of accepting and handling connections until the process exits. **To avoid the thundering herd problem, the accept call is wrapped around a process shared mutex created by the parent process**. Any thread in any process must acquire the lock on this mutex before accepting a connection. This ensures that at any time, only one thread is calling accept.
Thread also checks the number of remaining connections that can be made to a process; if that number is equal to zero, then the thread exits because the process now only needs to wait for the active clients to finish.
After each thread accepts a connection, it decrements the remaining connections that the process can handle. After it finishes handling a client, it increments the total number of clients that have been handled. **If the number of clients handled become equal to CPP. Then the thread causes the process to exit.**

Maintaining Client Information:

Information about all the clients is maintained in an array created in the shared memory. All the threads have access to shared memory. Read-write exclusivity is carried with the help of semaphores.

Operations:

1. Checking for an existing client:
    a. The thread waits for the number of writers on shared memory to become 0.
    b. Then it iterates through the array of client structures and compares if the name of the client entry in the array matches that of the required client name.
    c. Suppose such an entry is found. The p_id and t_id of that client are returned.
    d. Otherwise, -1 is returned, indicating there is no such client.

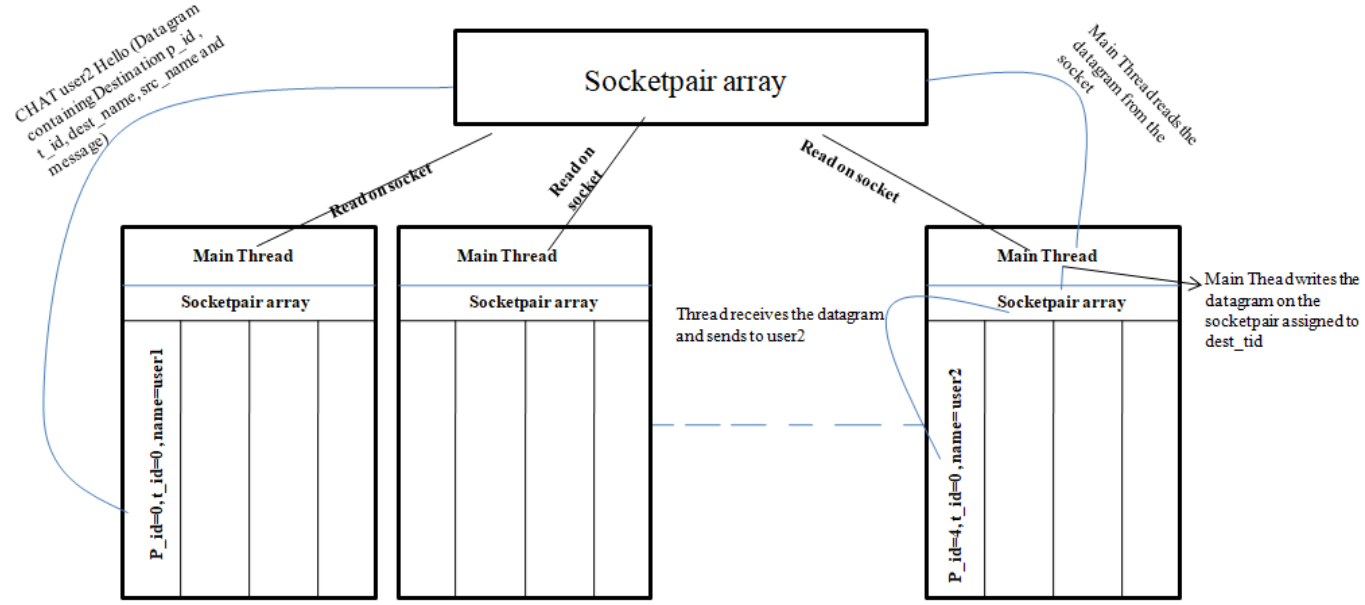
2. Inserting information about a new client.
    a. The thread waits for the number of readers and writers on shared memory to become 0 and increments the number of writers after it.
    b. The whole client entries array is traversed to check if the name that the client wants to assign itself is already in use.
    c. If it is not in use, the client information (p_id, t_id, name) is **stored in the first empty entry in the array.**

This operation is used when the client sends the JOIN <username> message to server.

3. Deleting information about a client:
    a. The thread waits for the number of readers and writers on shared memory to become 0 and increments the number of writers afterward.
    b. Client entries array is traversed to find the required entry to remove.
    c. The client entry is freed by setting all bytes in it to \0.

This operation is used when the client sends LEAV message to the server or when the client prematurely closes the connection before sending the LEAV message.

Inter Client Communication:



The parent process creates athe socket pair array, which is available to every process (let's call it sockarray_process). Each process has its own socket pair array meant to forward data to each thread (let's call this sockarray_thread).
If a user wants to chat with another user. It sends a message of the form CHAT <dest_username> <message>.
The thread handling this client completes this chat request by sending a datagram containing the chat message and other useful information to the thread handling the client with "dest_username."

The implementation of this functionality is as follows:

1. The thread searches for the dest_username in the shared memory and fetches the dest_pid and dest_tid.
2. It forms a datagram and sends it on the socket for process with p_id=dest_pid. The socket is accessed as sockarray_process[dest_pid][1].
3. The main thread of the process with p_id=dest_pid reads the datagram(on sockarray_process[dest_pid][0]), and with the help of dest_tid it forwards the datagram to the thread. (sockarray_thread[dest_tid][1]).
4. The destination thread reads the datagram and verifies if the destination name matches its client's name. It might be possible that the client with "dest_username" has left, and this thread is now handling a new user with a different username. In which case, the datagram is simply dropped.

5. After verification, the message content and the sender’s username are sent to the client with dest_username.