Internet Programming (Assignment 2)

Unmesh Joshi (VUnetID: uji300) and Koustubha Bhat (VUnetId: kbt350)

1st October 2012

Chapter 1

Assignment Documentation

1.1 Platforms

1 C Programs:

- gcc version 4.7.0 20120507 (Red Hat 4.7.0-5) (GCC)
- Thread model: posix
- Target: x86_64-redhat-linux

1.2 What to expect after *make*

A directory named **out** will be created in the directory from where make will be run. **out** will contain all C program executables.

1.3 Explanations of programs

1 serv1:
Requirements:

1. Server maintains a counter and this is a persistent variable.
2. Server listens to client requests.
3. Each client request is processed by incrementing the common persistent counter.
4. Server sends the incremented counter value back to the client.
5. The counter value 0 is the initial one and is never given to an actual client. Essentially counter value cannot be negative.

Explanation: We used file to maintain the counter. flock() allows us to have mutual exclusion between processes for accessing the counter.

2 serv2:

Requirements:

1. Server maintains a counter and this is a persistent variable.
2. Server listens to client requests.
3. **Each client request is processed by separate process.**
4. Server sends the incremented counter value back to the client.
5. The counter value 0 is the initial one and is never given to an actual client. Essentially counter value cannot be negative.

Explanation: Multiple processes may try to access and modify file containing counter simultaneously. File lock prevents this and thus maintains consistency of the file. Initially we thought of shared memory to maintain counters and semaphores for mutual exclusion, however maintaining counter across server crashes would not have been possible. Periodically committing this value to file would solve this problem. That is why we chose file to store counter. System call *flock()* gives us the desired effect of mutual exclusion.

3 serv3:

Requirements:

1. Server maintains a counter and this is a persistent variable.
2. Server listens to client requests.
3. **Each client request is processed by one of preforked processes.**
4. Server sends the incremented counter value back to the client.
5. The counter value 0 is the initial one and is never given to an actual client. Essentially counter value cannot be negative.

Explanation: We are creating 10 processes and every process waits for client connection. The main function (process) will wait for all children to die. Each child process blocks on *accept()* call which is surrounded by semaphore for mutual exclusion. This arrangement is done for the following purpose: When client request comes, only one server process should be signaled of this event. If we do not use semaphores around *accept()* call, then all processes would wait for clients, causing overhead of signalling each one by system, though only one succeeds to connect. [1]

The requirement is that clients should get the numbers as per their sequence of requests. In the preforked approach, any of the child processes that are executing accept() call can take the request and establish connection. Depending on when they acquire the lock of the file to be accessed (maybe competing with other child processes in doing so), to get and increment the counter, particular number will be sent (assigned) to the corresponding client. However, what is ultimately ensured is that no two clients receive the same number (of course until the counter overflows!). This concurrency problem can be explained in the context of "linearization " . In a given timeline, it suffices to prove the correctness of the implementation, if we can prove that in a uniprocessor setting, the client that received the lesser counter value came up prior to the one that received a higher value, without contradicting any laws of the environment. In this case, a server process incrementing the counter value would be a good linearization point.

4 client:
Requirements:

1. Resolve IP address of server.
2. Connect to server.
3. Request for counter.
4. Print the counter.

Explanation: getaddrinfo() function is used to fetch IP address from host name at client side. From the list of IP addresses returned by getaddrinfo(), we ignore "0.0.0.0" IP address. It is used for *resolving Domain Name* and hence cannot be expected to work for local hostnames or computer names. To test concurrent connections, we had shell script to repeatedly execute client program.

5 talk:
Requirements:

1. Server takes no parameter and starts listening on port.
2. Client takes server host name as parameter and has to connect to the same.
3. Multiple servers on same machine talking to their corresponding clients.
4. Use ncurses to separate the text written by the two parties.

Explanation: Client or server both have to read from console and read from network (socket) simultaneously. Hence we chose multithreading. One thread will read from console and write to the socket. Other thread will do vice a versa.

As both threads are using the one and only console output (one for reading, one for writing), it gives jumbled output because of interpenetration of texts. Thus we use ncurses library for separation of text areas.

We support multiple servers (on the same machine) connecting to their corresponding clients through separate ports for initial connection and actual communication. However, although networking part of this feature is working properly, due to ncurses library issues, message display on terminal is not working in case of multiple servers instances.

In order to close the chat session gracefully, we have made the server and the client agree upon a simple protocol as described below: When either of the parties chatting together, decides to end the session, he/she enters $\hat{D}$ ($\hat{D}$ + ENTER).

1.4 Answers to the questions

1 Answer to the Question A:
It is a one-process-per- request scenario. Persistence of counter is the only issue. As we create process only after *accepting* the connection, we do not need to have mutual exclusion for access to the counter file. We solved the persistence issue by storing counter in file. In case of concurrent requests by client, they will get served one after another.

2 Answer to the Question B:
The issues are Persistence of counter, Mutual exclusion for access to the counter file, waiting for all child processes and mutual exclusion for *accept()*. Apart from this, periodical regeneration of child processes to fill in the preforked process pool is also an issue. Use of file and *flock* will solve mutual exlcusion and persistence of counter. Use of semaphores around *accept* system call ensures mutual exclusion for *accept*, giving performance improvement. If we maintain count of died processes using signal handler, we can arrange for regeneration of processes. The server would behave correctly for concurrent requests because of the solutions discussed above.

3 Answer to the Question C:
One-process-per-client is the most appropriate server structure for this problem. As it is communication between only two parties, server will not talk to another client unless current client is disconnected.

4 Answer to the Question D:
Yes. We can have client and server executing simultaneously on the same machine. Two servers executing simultaneously on the same machine is also possible with one socket for connection and another for communication. So that

connection socket will be (again and again) resued by multiple servers. Using this approach, we can run SAME program to create multiple servers running on same machine simultaneously. Please refer another pdf for diagram.

5 Answer to the Question E:
We used Multithreading where one thread reads from console while the other reads from the network. To separate the text area of the two parties, ncurses library is used.

Bibliography

[1] Spyros Voulgaris, Lecture slides (Sockets), Fall 2012, Vrije Universiteit.