Network Programming, Fall, 2003

Project 2: Remote Working Ground (rwg)

9123571 Chung-Wei Hang
at Learning Lab, CIS, NCTU
gis91571@cis.nctu.edu.tw

2003.11.10

Requirment

In this homework, you are asked to design chat-like systems, called remote working systems (server only). In this system, users can meet/talk/make/work with friends. Basically, this system supports all functions, as you did in the first project. In addition, all clients can see what all on-line users are working.

Program Overview

This time, we have to implement two kinds of remote working ground servers, and both two kinds of servers have to support several new functions, including original functions in project 1:

- **who**
- **name**
- **yell**
- **tell**
- **list**
- **redirect and receive** [1]

[1] This function is not needed in concurrent connection-oriented paradigm.

Single-Process Concurrent Paradigm

The implementation of this part is the same as project 1, except several code segments in "rwgd.c":

- **struct client** This structure is responsible for all attributes of a single client, such as pipe, redirect and receive information, yell messages, and some other important attributes. This structure is added because in concurrent connection-oriented paradigm, each client is handled by a single child process. But in single-process concurrent paradigm, we have to handle this in a single process. Therefore, we need this structure to handle different information of many clients.

- **struct redirectnode** The "redirect node" is the structure which stores information of redirect function, including redirect pipes and redirect target. Each client has its own redirect list, which is consist of a large number of "redirect nodes". These nodes indicate that the redirect pipes redirected from other clients. For example, the 14th elements in "redirect-list" is the pipe which contains data sent by the client with 14 fd.

- **main** The "main" function uses "select" system function and "fd-set" to implement single-process concurrent paradigm by listening connection fd's.

- **rwg** This function is the same as the "ras" function in project 1, which "read" the client command. The "ras" function "read" the command with an infinite loop, but the "rwg" is not. Because the "rwg" is called only when there is something can be "read". After the command is executed, the connection is listened by "select", instead of the busy waiting way used in "ras".

- **parse** This part adds all new functions in project 2, such as "who", "name", "yell", ... We all discuss these new functions in "Implementation Issues" section.

Concurrent Connection-Oriented Paradigm

This part I am asked to implement in concurrent connection-oriented paradism:

- **struct client** This structure puts all necessary information in the shared memory, and processes can communicate each other via accessing this structure in the shared memory, including pid, connection status, messages, redirect and receive information, ...
- **struct shmdata** "shmdata" is consist of an array of "struct client", and it indicates all the clients on the server.
- **struct msg** "msg" struct takes care of message transmitted in the system.

Implementation Issues

Single-Process Concurrent Paradigm

I sumerize some important implementation issues used in this project as follow:

Client Management

The program maintains a client table as large as fd table, including all attributes of a client, such as pipe fd's, yell messages history, tell messages sent by other clients, redirect pipe fd's written by other clients. And clients can be identified by the socket fd's they use.

Talk Functions

These talk functions, including who, name, tell, yell and list, are implemented by common global object like the client table. It is easy to implement talk functions in single-process concurrent paradigm.

Redirect and Receive

This is the most important part in this paradigm. Since the receiver client may not receive data sent by sender immediately, the sender client cannot redirect another data pipe to the same receiver client util the receiver client reads the first redirect pipes. However, one client can send to different clients in the same time, and one client can be sent by many sender clients, too. Therefore, if there are n clients on-line, a client can have at most $n - 1$

redirect pipes at the same time. But there are some restrictions between redirect pipes and pipes in a single client. For example:

> ls |
> number >3 /* receive redirect pipe sent by client 3 */

The result of "ls |" will not be as input in "number <3". It will be as input in next command. But we can input as follow:

* client 1 *
> ls -l |
> cat >2
client 2 *
> number <2 >1

We can receive a redirect pipe and redirect to another client immediately. As a result, I summerize some features of "redirect and receive":

- If the receiver client doesn't receive the firse pipe redirected by the sender client, the sender client cannot redirect the second pipe to the same receiver client.
- The pipes in the single client and the redirect pipes cannot be input as the same command nor output as the same command at the same time.
- Except those cases above, the pipes can work correctly.

Concurrent Connection-Oriented Paradigm

In this paradigm, processes access the shared memory with "shmget", "shmat", and "shmdt". There are some important issues in my implementation:

Semaphore

I use a semaphore with id "semid" to ensure the mutural exclusive access of the shared memory. It means when some process would like to access the shared memory, it should call "sem-wait(semid)" first to wait other processes finish their action to the shared memory. After a process finish its job to the shared memory, it should need to call "sem-signal(semid)" to wake up the processes which is waiting to access the shared memory.

Signal SIGUSR1

Sometimes, when process A needs to print some messages to the client of process B, A could access the shared memory to tell what message it want to print. Then, A may need to use "signal" and "kill" function to signal B to print the message in the shared memory immediately. Therefore, I use "SIGUSR1" signal type and "printmsg" function to handle this kind of requirements.

FIFO

I use FIFO (named pipes) to handle "redirect" and "receive" functions in this project with concurrent connection-oriented paradigm. Instead of pipes with related processes, FIFO provides us a convenient way for non-related processes to communicate with each others. FIFO can be identified with its file name. Besides, processes can access the FIFO with "mkfifo", and "open" functions to get FIFO's fd. In this project, I use non-blocking read and write of FIFO's. Other implementation issues are similar with the cases in single-process concurrent paradigm, except the FIFO structures. In addition, because of both read and write need to be opened in some process at the same time with "non-blocking" read and write. I use another signal type "SIGUSR2" and "openreadfifo" functions to handle this, which is described as following.

Signal SIGUSR2

When using non-blocking read and write FIFO, it is necessary to get read and write fd at the same time. Because when we call "open" function, "read open" will wait some process to call "write open", vice versa. But the fd of read and write are always belong to different processes. Then, I use signal type "SIGUSR2" and "openreadfifo" function to form a process which call the "write open" to signal another processs to call the "read open". Thus, both read and write fd can be gotten in both processes.

Note

- The numbers of messages and clients are limited in this project, because they are implemented in arrays.
- Environment variables in processes are the same. It means when you

change one variable in one process, and the change will be shown in all processes.

- This document is powered by LaTeX.
- The time used in paradigm one and two are 3:2, and half of the time of the second paradigm is used to implement FIFO.