CSE 344

SYSTEM PROGRAMMING

SPRING 2023

MIDTERM

AHMET CEBECİ

1901042708

INTRODUCTION

This code is a server-client application in C that communicates through FIFOs, also known as named pipes. FIFOs are a method of Inter-Process Communication (IPC). IPC is a mechanism that allows processes to communicate with each other and synchronize their actions.

The server can handle multiple clients, up to a maximum defined in the code (5 in this case). Each client sends requests to the server through FIFOs, and the server responds to these requests accordingly. There are different types of requests, such as "quit", "killServer", and "list".

SYSTEM ARCHITECTURE

The system architecture for this code is a simple client-server model with the server handling multiple clients. The server is set up to accept commands from clients and execute them, then sending back the response to the corresponding client.

The communication mechanism used between the server and the clients is FIFOs (First In, First Out), a type of named pipe in Unix/Linux. Named pipes are special files that exist as a method for Inter-Process Communication (IPC), allowing data to flow between processes.

DESIGN DECISIONS

Multiprocessing vs Multithreading: The decision to use multithreading in the server's design enables it to efficiently handle multiple client connections concurrently. In contrast to multiprocessing, multithreading has less overhead since threads share the same memory space.

FIFOs for IPC: Using FIFOs for inter-process communication was another key decision. FIFOs, as named pipes, offer a simple and effective way to communicate between processes with little overhead. They are easily accessible through the file system, which simplifies the communication process between unrelated processes.

Request Structure: A custom request structure (struct request) was designed to encapsulate all necessary information for a single client request. This structure contains a process ID (pid) and a command.

Signal Handling: The server and client code includes a signal handler for dealing with interrupt signals (SIGINT). This allows for graceful termination of the server and client when receiving a SIGINT signal, typically triggered by the user pressing Ctrl+C.

IMPLEMENTATION DETAILS

Server Setup: The server begins by checking command-line arguments and creating a directory to hold logs. It then sets up a FIFO and opens it for reading and writing. The server then sets up a signal handler for the SIGINT signal.

Client Connection: When a client connects to the server, it writes its pid and its own FIFO's name to the server's FIFO. The server then opens the client's FIFO to establish a two-way communication channel.

Command Processing: The server processes client commands in a separate thread. It can handle different types of commands such as "quit", "killServer", and "list". When the command processing is complete, it closes the client's file descriptor and the thread exits.

Client Commands: The client reads user input and sends commands to the server. Depending on the command, the client may send a signal to the server, request a list of files from the server's directory, or disconnect from the server.

Logging: The server logs all client commands to individual log files in a server-specified directory. This is done with the write_server_log() function.

Signal Handling: If a SIGINT signal is received, the signal handlers in both the client and server will close the open file descriptors, remove their respective FIFOs, and then terminate.

SERVER CODE

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <signal.h>
#include <sys/wait.h>
#include <dirent.h>

#define MAX_CLIENTS 5

struct request {
    pid_t pid;
    char command[100];
};

char server_fifo[100];
int server_fd;
int dummy_fd;
int client_pids[MAX_CLIENTS];
int num_clients = 0;
struct request req;

char* dirname;

void signal_handler(int signum) {
    if (signum == SIGINT) {
        printf("\n>> kill signal from client.. terminating...\n");
        close(server_fd);
        unlink(server_fifo);
        printf(">> bye\n");
        exit(0);
    }
}

int client_already_connected(pid_t pid) {
    for (int i = 0; i < num_clients; i++) {
        if (client_pids[i] == pid) {
            return 1;
        }
    }
    return 0;
}

void write_server_log(pid_t client_pid, const char* message) {
    printf("%s\n", message);
    char log_file[200];
    sprintf(log_file, "%s/client_%d.log", dirname, client_pid);

    FILE* log = fopen(log_file, "a");
    if (log == NULL) {
        perror("fopen");
        return;
    }

    fprintf(log, "%s\n", message);
    fclose(log);
}
```

```c
void *process_client_request(void *client_fd_ptr) {
    int client_fd = *(int *)client_fd_ptr;
    free(client_fd_ptr);

    struct request req;
    pid_t client_pid = -1;

    ssize_t num_bytes;
    while ((num_bytes = read(client_fd, &req, sizeof(struct request))) > 0) {
        client_pid = req.pid;

        char log_message[200];
        sprintf(log_message, "Client %d command: %s", client_pid, req.command);
        write_server_log(client_pid, log_message);

        if (strcmp(req.command, "quit") == 0) {
            printf(">> Client PID %d has quit\n", client_pid);
            break;
        }

        else if (strcmp(req.command, "killServer") == 0) {
                    raise(SIGINT);
                }

        else if (strcmp(req.command, "list") == 0) {
            DIR *dir = opendir(dirname);
            struct dirent *entry;
            while ((entry = readdir(dir)) != NULL) {
                write(client_fd, entry->d_name, strlen(entry->d_name) + 1);
            }
            closedir(dir);
        }
    }
    if(num_bytes <= 0 && (client_pid != -1 && strcmp(req.command, "quit") != 0)) {
        printf(">> Client PID %d disconnected\n", client_pid);
    }

    close(client_fd);
    pthread_exit(NULL);
}
```

```c
int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: server <dirname> <max_clients>\n");
        exit(1);
    }

    dirname = argv[1];
    struct stat st = {0};
    if (stat(dirname, &st) == -1) {
        mkdir(dirname, 0700);
    }

    printf(">> Server Started PID %d…\n", getpid());
    printf(">> waiting for clients...\n");


    sprintf(server_fifo, "/tmp/server.%d.fifo", getpid());
    if (mkfifo(server_fifo, 0666) < 0) {
        perror("mkfifo");
        exit(1);
    }

    server_fd = open(server_fifo, O_RDWR);
    if (server_fd < 0) {
        perror("open");
        exit(1);
    }

    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while (1) {
        ssize_t num_bytes = read(server_fd, &req, sizeof(struct request));
        if (num_bytes <= 0) {
            perror("read from client");
            exit(1);
        }

        if (client_already_connected(req.pid)) {
            continue;
        }

        if (num_clients >= MAX_CLIENTS) {
            printf(">> Maximum number of clients (%d) has been reached. Cannot accept more clients.\n", MAX_CLIENTS);
            exit(1);
        }

        printf(">> Client PID %d connected as "client%d"\n", req.pid, num_clients + 1);
        client_pids[num_clients++] = req.pid;

        int *client_fd_ptr = malloc(sizeof(int));
        *client_fd_ptr = open(req.command, O_WRONLY);
        if (*client_fd_ptr < 0) {
            perror("open");
            continue;
        }

        pthread_t thread;
        if (pthread_create(&thread, NULL, process_client_request, client_fd_ptr) != 0) {
            perror("pthread_create");
            continue;
        }
    }
```

First, the server code defines the request structure and some global variables. The request structure includes a process ID (pid) and a command. Global variables include an array for client pids, a count of the number of clients, and the request structure itself.

The server code then defines a few functions:

signal_handler(): This function handles SIGINT signals (interrupt signals, usually triggered by the user pressing Ctrl+C). When the server receives a SIGINT signal, it will close the server file descriptor, remove the server's FIFO, and then exit.

client_already_connected(): This function checks if a client is already connected to the server.

write_server_log(): This function logs messages from clients in a log file.

process_client_request(): This function processes requests from clients. If a client sends a "quit" command, it will break the loop and end the client connection. If a client sends a "killServer" command, it will raise a SIGINT signal to stop the server. If a client sends a "list" command, it will list all files in the server's directory.

main(): This function is the entry point of the server code. It checks command-line arguments, initializes the server, sets up the signal handler for SIGINT, and then enters a loop where it waits for clients to connect and sends client requests to be processed in

separate threads.

CLIENT CODE

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <signal.h>
#include <pthread.h>

struct request {
    pid_t pid;
    char command[100];
};

struct request req;
char server_fifo[100];
char client_fifo[100];
int server_fd;
int client_fd;

void signal_handler(int signum) {
    if (signum == SIGINT) {
        printf("\n>> terminating client...\n");
        close(server_fd);
        close(client_fd);
        unlink(client_fifo);
        printf(">> bye\n");
        exit(0);
    }
}

void* connect_server(void* args){
    server_fd = open(server_fifo, O_WRONLY);
    if (server_fd < 0) {
        perror("open");
        exit(1);
    }

    strcpy(req.command, client_fifo);
    write(server_fd, &req, sizeof(struct request));

    client_fd = open(client_fifo, O_RDONLY);
    if (client_fd < 0) {
        perror("open");
        exit(1);
    }
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: client Connect <server_pid>\n");
        exit(1);
    }

    if (strcmp(argv[1], "Connect") != 0) {
        fprintf(stderr, "The first argument should be 'Connect'\n");
        exit(1);
    }
```

```c
    if (mkfifo(client_fifo, 0666) < 0) {
        perror("mkfifo");
        exit(1);
    }

    pthread_t connect_thread;
    pthread_create(&connect_thread, NULL, connect_server, NULL);

    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while (1) {
        printf("enter command: ");
        fgets(req.command, sizeof(req.command), stdin);
        req.command[strcspn(req.command, "\n")] = '\0';


        if (strcmp(req.command, "help") == 0) {
            printf("Possible commands:\n");
            printf("quit - disconnect from the server\n");
            printf("killServer - send a kill signal to the server\n");
            printf("list - list files in server's directory\n");
        }

        else if (strcmp(req.command, "quit") == 0) {
            printf(">> Client PID %d is quitting...\n", req.pid);
            close(server_fd);
            close(client_fd);
            unlink(client_fifo);
            break;
        }
        else if (strcmp(req.command, "killServer") == 0) {
            write(server_fd, &req, sizeof(struct request));
            close(client_fd);
            close(server_fd);
            unlink(client_fifo);
            break;
        }
        else if (strcmp(req.command, "list") == 0) {
            write(server_fd, &req, sizeof(struct request));
            char file_list[1024];
            read(client_fd, file_list, sizeof(file_list));
            printf("Files in server's directory:\n%s\n", file_list);
        }
        else {
            write(server_fd, req.command, strlen(req.command) + 1);
        }
        char buf[100];
        while (read(client_fd, buf, sizeof(buf)) > 0) {
            printf("%s", buf);
        }
    }
```

The client code is somewhat similar to the server code. It also defines the request structure and some global variables.

It then defines two functions:

signal_handler(): This function works similarly to the server's signal handler, except it's for the client. When the client receives a SIGINT signal, it will close both the server and client file descriptors, remove the client's FIFO, and then exit.

connect_server(): This function connects to the server by opening the server's FIFO and writing the client's pid and FIFO to the server.

main(): This function is the entry point of the client code. It checks command-line arguments, creates a new thread to connect to the server, sets up the signal handler for SIGINT, and then enters a loop where it waits for user input to send to the server.

OUTPUT

Connect:

```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/midterm$ make
gcc -o server server.c -lpthread -lrt
gcc -o client client.c -lpthread -lrt
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/midterm$ ./server mydir 5
>> Server Started PID 2025…
>> waiting for clients...
>> Client PID 2026 connected as “client1”
>> Client PID 2029 connected as “client2”
>> Client PID 2032 connected as “client3”
>> Client PID 2034 connected as “client4”
>> Client PID 2036 connected as “client5”
>> Max clients reached. Ignoring client PID 2038
```

Help:

```
>> Enter command: help
Possible commands:
quit - disconnect from the server
killServer - send a kill signal to the server
list - list files in server's directory
>> Enter command:
```

List:

```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/hw4/Cebeci_Ahmet_1901042708$ ./client Connect 7365
enter command: list
Files in server's directory:

enter command:
```

Quit:

```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/hw4/Cebeci_Ahmet_1901042708$ ./client Connect 7340
enter command: quit
>> Client PID 7356 is quitting...
```

Kill Signal by CTRL+C:

```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/midterm$ ./server mydir 5
>> Server Started PID 2040…
>> waiting for clients...
>> Client PID 2041 connected as “client1”
^C
>> kill signal from client.. terminating...
>> bye

>> kill signal from client.. terminating...
>> bye
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/midterm$
```