CSE 344

SYSTEM PROGRAMMING

SPRING 2023

MIDTERM

AHMET CEBECİ

1901042708

INTRODUCTION

This project constitutes a simple inter-process communication system using named pipes (or FIFOs) in a UNIX or Linux environment. This system consists of a server and a client. The server receives requests from multiple clients, processes them, and then sends back responses. The client sends requests to the server and waits for responses. The server can handle multiple clients at once.

SYSTEM ARCHITECTURE

The system is designed as a client-server model where multiple clients can interact with a single server concurrently. Communication between the client and the server is achieved using named pipes (also known as FIFOs), which are a feature provided by the UNIX/Linux operating system.

The server creates a named pipe which clients use to send requests. Each client also creates a named pipe which the server uses to send responses. This two-way communication allows for interaction between the client and server.

DESIGN DECISIONS

The design decision to use named pipes for interprocess communication was primarily influenced by their ease of use and support in the UNIX/Linux operating system. Named pipes allow for simple, bidirectional communication between

processes. They are identified by a filename in the filesystem, which makes them easily accessible by any process that knows the name.

The server is designed to handle multiple clients at the same time. This is achieved by creating a new child process for each connected client. This allows the server to process multiple client requests concurrently, improving system throughput.

The client sends commands to the server in the form of text strings. This makes the system flexible and easy to extend with new commands.

IMPLEMENTATION DETAILS

The server code starts by creating a named pipe (server FIFO) and setting up a signal handler for SIGINT. It then enters a loop where it waits for client connections and reads client requests from the server FIFO. For each client request, the server creates a new child process to handle the client's command.

The client code starts by creating its own named pipe (client FIFO) and sending a request to the server that includes its PID. It then enters a loop where it waits for user to input commands. Depending on the input command, the client sends the appropriate request to the server and waits for a response.

The server and client use the read() and write() system calls to send and receive data through the named pipes. The server and client also use the open(), close(), and unlink() system calls to manage the named pipes.

The client supports three commands: quit, killServer, and list. The quit command disconnects the client from the server. The killServer command stops the server. The list command asks the server to send a list of files in the server's directory. The server reads these commands from the client's request, performs the requested operation, and sends a response back to the client.

SERVER CODE

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


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

    char* dirname = argv[1];
    int max_clients = atoi(argv[2]);

    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_RDONLY);
    if (server_fd < 0) {
        perror("open");
        exit(1);
    }

```

```
    server_fd = open(server_fifo, O_RDONLY);
    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) < 0) {
        perror("sigaction");
        exit(1);
    }

    while (1) {
        pid_t finished_pid;
        int status;
        while ((finished_pid = waitpid(-1, &status, WNOHANG)) > 0) {
            // A child process has finished
            for (int i = 0; i < num_clients; ++i) {
                if (client_pids[i] == finished_pid) {
                    printf(">> Client PID %d disconnected\n", finished_pid);
                    // Remove the disconnected client PID from the list.
                    for (int j = i; j < num_clients - 1; ++j) {
                        client_pids[j] = client_pids[j + 1];
                    }
                    --num_clients;
                    break;  // Break the loop to avoid creating a new client connection.
                }
            }
        }

        ssize_t num_bytes = read(server_fd, &req, sizeof(struct request));
        if (num_bytes <= 0) {
            perror("read");
            continue;
        }

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

        if (num_clients >= MAX_CLIENTS) {
            printf(">> Max clients reached. Ignoring client PID %d\n", req.pid);
            continue;
        }

        int client_pid = req.pid;
        printf(">> Client PID %d connected as “client%d”\n", client_pid, num_clients+1);

        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        }
```

```
    else if (pid == 0) {
        // Child process
        char client_fifo[100];
        sprintf(client_fifo, "/tmp/client.%d.fifo", client_pid);
        int client_fd = open(client_fifo, O_RDWR);
        if (client_fd < 0) {
            printf(">> Client PID %d not found\n", client_pid);
            exit(1);
        }

        while (1) {
            // Read the command from the client
            ssize_t num_bytes = read(client_fd, &req, sizeof(struct request));
            if (num_bytes <= 0) {
                printf(">> Client PID %d disconnected\n", client_pid);
                _exit(0);  // Notify the parent process that this child process has exited.
            }

            if (strcmp(req.command, "quit") == 0) {
                close(client_fd);
                exit(0);  // Notify the parent process that this child process has exited.
            }
            else if (strcmp(req.command, "killServer") == 0) {
                raise(SIGINT);
            }
            else if (strcmp(req.command, "list") == 0) {
                char file_list[1024] = "";  // Make sure this is large enough
                DIR *d;
                struct dirent *dir;
                d = opendir(dirname);
                if (d) {
                    while ((dir = readdir(d)) != NULL) {
                        strcat(file_list, dir->d_name);
                        strcat(file_list, "\n");
                    }
                    closedir(d);
                }
                write(client_fd, file_list, strlen(file_list) + 1);
            }
        }
    }
    else {
        // Parent process
        client_pids[num_clients] = pid;
        ++num_clients;
    }

}
return 0;
}
```

The server creates a named pipe (FIFO), then waits for clients to connect and send requests. It handles each client request in a child process, allowing it to serve multiple clients concurrently.

Here are the key steps performed by the server:

-The server starts and takes two arguments: a directory name and the maximum number of clients it can handle.

-The server creates a named pipe (FIFO) which is used for communication with the clients. This FIFO is located in the /tmp directory and named as server.PID.fifo, where PID is the process ID of the server.

-The server sets up a signal handler for SIGINT (the interrupt signal). If the server receives this signal, it cleans up by closing the FIFO and then exits.

-The server enters a loop where it waits for client connections. It reads client requests from the server FIFO. Each request includes the client's PID and a command.

-If the server has reached the maximum number of clients, it ignores any new incoming connections.

-For each client request, the server creates a new child process using fork(). In this child process, it opens the client's FIFO (which is expected to be named client.PID.fifo, where PID is the client's process ID), reads the client's command, and performs the requested operation. The server supports three commands: quit (to disconnect the client), killServer (to stop the server), and list (to list the files in the server's directory).

CLIENT CODE

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

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

struct response {
    int seqNum;
};

char server_fifo[100];
char client_fifo[100];
struct request req;
struct response resp;



int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: client <Connect/tryConnect> <serverPID>\n");
        exit(1);
    }
    char* connection_type = argv[1];
    int server_pid = atoi(argv[2]);

    sprintf(server_fifo, "/tmp/server.%d.fifo", server_pid);
    sprintf(client_fifo, "/tmp/client.%d.fifo", getpid());

    mkfifo(client_fifo, 0666);

    int server_fd = open(server_fifo, O_WRONLY);
    if (server_fd == -1) {
        perror("open");
        exit(1);
    }

    req.pid = getpid();
    write(server_fd, &req, sizeof(struct request));

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

    char command[100];
    while (1) {
        printf(">> Enter command: ");
        fgets(command, sizeof(command), stdin);
        command[strcspn(command, "\n")] = 0;
        strcpy(req.command, command);
```

```c
    while (1) {
        printf(">> Enter command: ");
        fgets(command, sizeof(command), stdin);
        command[strcspn(command, "\n")] = 0;
        strcpy(req.command, command);

        if (strcmp(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(command, "quit") == 0) {
            write(server_fd, &req, sizeof(struct request));
            close(client_fd);
            close(server_fd);
            unlink(client_fifo);
            break;
        } else if (strcmp(command, "killServer") == 0) {
            write(server_fd, &req, sizeof(struct request));
            close(client_fd);
            close(server_fd);
            unlink(client_fifo);
            break;
        } else if (strcmp(command, "list") == 0) {
            write(server_fd, &req, sizeof(struct request));
            char file_list[1024];  // Make sure this is large enough
            read(client_fd, file_list, sizeof(file_list));
            printf("Files in server's directory:\n%s\n", file_list);
        } else {
            write(server_fd, command, strlen(command) + 1);
        }
    }
    close(client_fd);
    close(server_fd);
    unlink(client_fifo);

    return 0;
}
```

The client sends requests to the server and waits for responses. It takes two arguments: the connection type and the server's PID.

Here are the key steps performed by the client:

-The client starts, takes the server's PID as an argument, and constructs the server's FIFO name.

-The client creates its own named pipe in the /tmp directory, named client.PID.fifo, where PID is its process ID.

-The client opens the server's FIFO in write mode, and sends a request that includes its PID.

-The client opens its own FIFO in read mode, then enters a loop where it waits for the user to input commands.

-Depending on the entered command, the client will perform different actions. If the command is quit, it will disconnect from the server. If the command is killServer, it will ask the server to stop. If the command is list, it will ask the server to send a list of files in the server's directory. It reads the server's response from its FIFO.

-After each command, the client closes its FIFO, unlinks it, and then exits.

In summary, this system demonstrates how to implement a multi-client server using named pipes in a UNIX/Linux environment. The server can handle multiple clients concurrently, each in its own child process. Both the server and the client use FIFOs for communication. The client sends requests to the server, and the server sends responses back to the client.

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:
```

Quit:

```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/midterm$ ./client Connect 2025
>> Enter command: quit
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/SystemHomework/midterm$
```

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$
```