CSE344 – System Programming

HW4 – Report

Ömer Faruk Bitikçioğlu

161044010

Client

The code begins by including necessary libraries and header files. These provide various functionalities required for the program.

Next, there are functions that facilitate communication with the server. The sendRequest function writes a request to the FIFO channel opened for communication. It takes the file descriptor of the FIFO and the request structure as inputs. If there is an error during the write operation, an error message is displayed, and the program exits.

The getResponse function reads the response from the FIFO channel. It dynamically allocates memory for the response structure and reads the response using the read system call. If there is an error during the read operation, an error message is displayed, the allocated memory is freed, and the program exits. The response structure is then returned.

There is also a signal handler function, signintHandler, which handles the SIGINT signal triggered when the user presses Ctrl+C. This function is responsible for cleaning up the client-side FIFOs and terminating the process. It unlinks (removes) the client FIFOs and terminates the process.

The connectServer function sends a connection request to the server. It takes the client structure, the server FIFO name, and the request type as inputs. It allocates memory for the request structure and sets the client PID and command based on the request type. The server FIFO is opened in write-only mode, and if the open operation fails, an error message is displayed, and the program exits. The request is then sent using the sendRequest function, and the allocated memory is freed. Finally, the server FIFO is closed.

The connectionEstablished function is called when the connection to the server is successfully established. It continuously reads user input commands from the standard input and sends them to the server for execution. It allocates memory for the request structure, sets the client PID and command based on the user input, and opens the server FIFO in write-only mode. If the open operation fails, an error message is displayed, and the program exits. The request is sent using the sendRequest function, and the allocated memory is freed. The client response FIFO is opened in read-only mode, and the response from the server is obtained using the getResponse function. The response is then printed to the standard output. If the response indicates that the client should quit or the server is terminating, the loop ends. Finally, the allocated memory is freed, and the client response FIFO is closed.

The main function serves as the entry point of the client-side code. It validates the command-line arguments, sets up a signal handler for Ctrl+C, and checks the existence of the server process and its FIFO file. It initializes the client structure with the client PID and creates the client request and response FIFOs. A connection request is sent to the server using the connectServer function, and the connection response is obtained. Depending on the response code, either the connectionEstablished function is called to interact with the file system or the program waits for the queue from the server. After the interaction or waiting is finished, the client FIFOs are cleaned up, and the program terminates.

Server

The code begins by including necessary header files and defining some constants and data structures. It also declares a global variable called "terminate_flag" that is used to control the termination of the threads.

Next, there are some function prototypes and a signal handler for the SIGINT signal, which is triggered by pressing Ctrl+C. The signal handler is responsible for cleaning up resources and terminating the server.

The code also includes a function called "getFormattedDateTime" that returns the current date and time in a formatted string.

There are several functions defined in the code to process different types of client requests. For example, there is a function called "processList" that lists the files in a specified directory, a function called "processReadFile" that reads the content of a file (either the entire file or a specific line), and a function called "processWriteToFile" that writes a string to a file (either at the end of the file or at a specific line).

The code also includes a function called "serveClients" that is executed in separate threads to serve client requests. Inside this function, the server receives a request from a client, processes the request, prepares a response, and sends the response back to the client.

The "main" function is the entry point of the program. It starts by checking the command-line arguments and setting up the signal handler. Then, it initializes data structures and variables required for the server.

The code creates a directory (if it doesn't exist) for storing server files and opens a log file for writing server logs. It also creates a FIFO (named pipe) for communication with clients.

After setting up the server, it enters a loop where it waits for client connections and handles them. The server maintains an array of connected clients, a queue of waiting clients, and queue of client requests.

Threads in thread pool waits for new client requests, wake up when the request queue is filled, handle the request and return the value to the requesting client.

The main thread of the server continuously waits for new client connections and add them to the connected clients array until the maximum number of connections is reached. When a client connects, the server gets client requests, add them into a request queue and signal the threads in the thread pool to deal with these requests.

The server continues this process until it receives the “killServer” request or SIGINT signal, indicating a termination request. When the signal is received, the server cleans up resources, closes the log file, and terminates the program.

Client Queue (client_queue.h)

This code implements a client queue data structure. It allows you to store client objects in a queue-like manner, where you can add clients to the end of the queue and remove clients from the front of the queue.

The code defines two structs: client_node_t and client_queue_t. client_node_t represents a node in the client queue. It contains a pointer to a client_t object, which represents a client, and a pointer to the next node in the queue.

client_queue_t represents the client queue itself. It has two pointers, head and tail, which keep track of the first and last nodes in the queue, respectively.

To use the client queue, you can create an empty queue using the createClientQueue function. This function allocates memory for the queue structure and initializes the head and tail pointers to NULL.

You can add a client to the queue using the enqueueClient function. It takes a client queue and a client object as parameters. This function creates a new node, assigns the client object to it, and updates the tail pointer to point to the new node. This effectively adds the client to the end of the queue.

To remove and retrieve the first client from the queue, you can use the dequeueClient function. It takes a client queue as a parameter and returns the first client in the queue. It checks if the queue is empty, and if not, it removes the first node, updates the head pointer to point to the next node, and returns the client object.

The isClientInQueue function allows you to check if a client with a specific process ID exists in the queue. It takes a client queue and a client object as parameters. It iterates through the nodes in the queue, compares the process ID of each client with the given process ID, and returns 1 if a matching client is found or 0 if not.

Finally, the freeClientQueue function frees the memory allocated by the client queue. It iterates through the nodes in the queue, frees the client objects, frees the nodes themselves, and then frees the queue structure.

Client.h

The header file defines a struct called client_t, representing a client in the system. It has the following fields:

- int id: An identifier for the client.
- pid_t pid: The process ID (PID) of the client.
- char *fifo_req_name: A pointer to a character array representing the name of the client's read FIFO (First In, First Out) pipe.
- char *fifo_res_name: A pointer to a character array representing the name of the client's write FIFO pipe.

The createClient function is also defined in this header file. It takes two parameters: pid_t cl_pid, which represents the process ID of the client to be created, and int cl_count, which is the count of clients created so far. The function dynamically allocates memory for a client_t object and initializes its fields based on the provided arguments.

Inside the function, memory is allocated for the client_t structure using malloc. If memory allocation is successful (cl is not NULL), the fields of the structure are populated: id is set to cl_count, pid is set to cl_pid.

Two more character arrays, fifo_req_name and fifo_res_name, are allocated using malloc to store the names of the client's read and write FIFO pipes. These names are generated using the snprintf function and specific templates (CLIENT_FIFO_REQ_TEMPLATE and CLIENT_FIFO_RES_TEMPLATE) along with the client's process ID.

Finally, the created client_t object is returned from the function.

Bibo_cl_sv.h

This code defines a header file named "bibo_cl_sv.h" that contains various definitions and constants related to the client-server communication in a system.

CLIENT_FIFO_REQ_TEMPLATE: A template string representing the path pattern for the client's request FIFO (First In, First Out) pipe. It contains a placeholder %d that will be replaced with the client's process ID.

CLIENT_FIFO_RES_TEMPLATE: A template string representing the path pattern for the client's response FIFO pipe. It also contains a placeholder %d for the client's process ID.

CLIENT_FIFO_NAME_LEN: The length of the character array needed to store the client's FIFO pipe names. It is calculated based on the length of the template strings and an additional space for the process ID.

SERVER_FIFO_TEMPLATE: A template string representing the path pattern for the server's FIFO pipe. It contains a placeholder %d for the server's process ID.

SERVER_FIFO_NAME_LEN: The length of the character array needed to store the server's FIFO pipe name. It is calculated based on the length of the template string and an additional space for the process ID.

SERVER_LOG_TEMPLATE: A template string representing the log file name pattern for the server. It contains a placeholder %d for the server's process ID.

SERVER_LOG_NAME_LEN: The length of the character array needed to store the server's log file name. It is calculated based on the length of the template string and an additional space for the process ID.

MAX_RESPONSE_LEN: The maximum length of the response message that can be sent from the server to the client. It is set to 1024 bytes (or 1 kilobyte).

The header file also defines two structs:

request_t: Represents a client request. It has two fields:

- char cmd[100]: A character array to store the command string of the request.
- pid_t cl_pid: The process ID of the client.

response_t: Represents a server response. It has two fields:

- char msg[MAX_RESPONSE_LEN]: A character array to store the response message.

- response_code_e code: An enumeration value representing the response code, which can be one of the following:
    - OK: Indicates a successful response.
    - WAIT: Indicates a response that requires the client to wait.
    - FAIL: Indicates a failed response.

These structures are used to exchange data and information between the client and the server in the system.

Request_queue.h

This code defines a header file named "request_queue.h" that contains structures and functions related to a queue for storing client requests.

The header file defines two structures:

cl_req_node_t: Represents a node in the client request queue. It has two fields:

- request_t *cl_req: A pointer to the client request.
- struct cl_req_node_t *next: A pointer to the next node in the queue.

cl_req_queue_t: Represents the client request queue. It has two fields:

- cl_req_node_t* head: A pointer to the head (first node) of the queue.
- cl_req_node_t* tail: A pointer to the tail (last node) of the queue.

The header file provides several functions to operate on the client request queue:

createClientRequestQueue(): Creates an empty client request queue and returns a pointer to it.

enqueueClientRequest(cl_req_queue_t *q, request_t *cl_req): Adds a new client request to the tail of the queue. It takes a pointer to the queue and a pointer to the client request as parameters.

dequeueClientRequest(cl_req_queue_t *q): Removes and returns the first client request from the head of the queue. It takes a pointer to the queue as a parameter and returns a pointer to the dequeued client request. If the queue is empty, it returns NULL.

freeClientRequestQueue(cl_req_queue_t *q): Frees the memory occupied by the client request queue. It takes a pointer to the queue as a parameter and frees all the nodes and requests in the queue.