**—------------------------------------------------------------------------------------------------------------------------------------**

**Node.c Code Explained:**

This is a C program that starts a server and a client program as child processes, using pipes to communicate between them. The code can be divided into several sections:

**1. Header files:**

The program includes several header files:

stdio.h: Provides input/output functionality for the program.

netdb.h: Contains definitions for network database operations.

sys/socket.h: Provides definitions for socket-related functions.

sys/types.h: Defines various data types used in system calls.

unistd.h: Defines various symbolic constants and types, and declares miscellaneous functions.

**2. Variable declarations:**

The program declares several integer arrays to represent pipes that will be used for communication between the parent process and the child processes. There are two sets of pipes:

serverpipe1 and serverpipe2, which the server process will use to communicate with the parent process.

clientpipe1 and clientpipe2, which the client process will use to communicate with the parent process.

**3. Pipe creation:**

The program creates the four pipes using the pipe() function. The pipe() function creates a pair of connected sockets (pipes): one socket is for reading and the other is for writing.

**4. Child process creation:**

The program creates two child processes using the fork() function. The first child process is responsible for running the server program, while the second child process runs the client program.

The fork() function creates a new process by duplicating the calling process. The new process, called the child process, is an exact copy of the original process (the parent process), except for its process ID and a few other attributes.

**5. Server child process:**

The server child process retrieves the file descriptors of the two pipes it will use to communicate with the parent process, by converting them from integers to strings using the sprintf() function.

The server child process then uses the execl() function to replace itself with the server program. The execl() function replaces the current process image with a new process image specified by the path name passed as the first argument, and passes the remaining arguments to the new process.

**6. Client child process:**

The client child process retrieves the file descriptors of the two pipes it will use to communicate with the parent process, by converting them from integers to strings using the sprintf() function.

The client child process then uses the execl() function to replace itself with the client program, passing the pipe file descriptors as arguments.

**7. Wait for child processes:**

The parent process waits for the two child processes to finish by calling the wait() function twice. The wait() function suspends the calling process until one of its child processes terminates.

The status of the terminated child process is stored in the integer variable passed to wait(). The process ID of the terminated child process is returned by wait().

**Variables Of Node.c Explained**

1. **serverpipe1[2]:** An integer array of length 2 used to represent a pipe between the server process and the parent process, where the server process writes and the parent process reads.

2. **serverpipe2[2]:** An integer array of length 2 used to represent a pipe between the server process and the parent process, where the parent process read and the server process reads.

3. **clientpipe1[2]:** An integer array of length 2 used to represent a pipe between the client process and the parent process, where the client process writes and the parent process reads.

4. **clientpipe2[2]:** An integer array of length 2 used to represent a pipe between the client process and the parent process, where the parent process read and the client process reads.

5. **t**: An integer variable used to store user input. It is used to decide whether to start the client process or not.

6. **statusServer:** An integer variable used to store the exit status of the server child process after it terminates.

7. **statusClient:** An integer variable used to store the exit status of the client child process after it terminates.

8. **pid1**: A variable of type pid\_t used to store the process ID of the terminated child process (either the server or the client process).

9. **pid2**: A variable of type pid\_t used to store the process ID of the other terminated child process (either the server or the client process).

10. **spipe1[5]:** A character array of length 5 used to store the string representation of the file descriptor of the write end of serverpipe1.

11. **spipe2[5]:** A character array of length 5 used to store the string representation of the file descriptor of the read end of serverpipe2.

12. **cpipe1[5]:** A character array of length 5 used to store the string representation of the file descriptor of the write end of clientpipe1.

13. **cpipe2[5]:** A character array of length 5 used to store the string representation of the file descriptor of the read end of clientpipe2.

**Functions Explained Of Node.c :**

1. **pipe(int pipefd[2]):** This function creates a pipe and returns two file descriptors pipefd[0] and pipefd[1], where pipefd[0] is for reading and pipefd[1] is for writing. The parameter pipefd is an integer array of length 2 used to store the two file descriptors.

2. **fork():** This function creates a new child process by duplicating the calling process. The child process is an exact copy of the parent process except for a few differences, such as the process ID and the return value of the fork() function. It returns 0 in the child process and the process ID of the child process in the parent process.

3. **sprintf(char \*str, const char \*format, ...):** This function writes a formatted string to the character array str. The second parameter format is a string that specifies the format of the output, and the additional parameters are the values to be formatted. In this program, it is used to convert the integer file descriptors to string representation.

4. **execl(const char \*path, const char \*arg0, const char \*arg1, ..., NULL):** This function replaces the current process image with a new process image specified by the file path. The first parameter path is a string that specifies the path of the executable file, and the remaining parameters are the command-line arguments passed to the new process. The list of arguments must be terminated by a NULL pointer. In this program, it is used to start the server and client processes.

5. **wait(int \*status):** This function suspends the calling process until one of its child processes terminates. If status is not NULL, it stores the exit status of the terminated child process. It returns the process ID of the terminated child process. In this program, it is used to wait for the termination of the server and client processes and get their exit status.

6. **printf(const char \*format, ...):** This function writes a formatted string to the standard output stream. The first parameter format is a string that specifies the format of the output, and the additional parameters are the values to be formatted. In this program, it is used to prompt the user to input a value.

7. **scanf(const char \*format, ...):** This function reads input from the standard input stream and stores the values in the specified variables. The first parameter format is a string that specifies the format of the input, and the additional parameters are the addresses of the variables where the input values are stored. In this program, it is used to get user input.

**—------------------------------------------------------------------------------------------------------------------------------------**

**Client.c Code Explained:**

This code is a C program that creates a client that connects to a server over a network. It includes a set of functions that perform specific tasks related to communication with the server. Here is an overview of the code:

1. The code includes several header files that define various functions and data types used in the program.

2. The code defines several global variables, including a serverIP address, an IP address for the client, and two file descriptors to read and write to a pipe.

3. The code defines several structures, including a request structure to send and receive data, a nodeList structure to hold a list of nodes, an IPC structure to facilitate interprocess communication, and a manifest structure to hold information about the files the client needs to retrieve from the server.

4. The code defines several functions to perform specific tasks:

**getIP:** retrieves the client's IP address and stores it in a global variable.

**communicate:** the main function that handles the client's interaction with the server. It first sends the client's IP address to the server, then retrieves a manifest from the server that contains information about the files the client needs to retrieve. Finally, it retrieves the files from the server.

**sendIP:** sends the client's IP address to the server using a request structure.

**getValidNodes:** retrieves a list of valid nodes from the server and sends it to the parent process using an IPC structure.

**getManifestData:** retrieves a manifest from the server and stores it in a manifest structure.

**getChunkFiles:** retrieves the files listed in the manifest from the server and stores them on the client's machine.

**sendNodeListToParent:** sends a list of valid nodes to the parent process using an IPC structure.

**clientSegmentForChunk:** retrieves a file from the server using a request structure and stores it on the client's machine.

5. The code uses socket programming to establish a connection between the client and the server. It uses the TCP protocol to transfer data.

6. The code includes several print statements that provide information about the program's progress and status.

7. The code also includes several comments that provide additional information about the program's functionality.

**Client.c Variables Explained:**

1. **serverIP[15]** - a character array to store the IP address of the server.

2. **ipaddr[15]** - a character array to store the IP address of the client.

3. **readPipe** - an integer to store the file descriptor of the read end of the pipe.

4. **writePipe** - an integer to store the file descriptor of the write end of the pipe.

5. **struct request** - a structure to send and receive data between client and server. It contains three members:

**type[1]** - a character array to represent the type of the request.

6. **data[1024]** - a character array to store the data being sent or received.

7. **filename[20]** - a character array to store the name of the file being transferred.

10. **save** - an integer to represent whether to save the file or not.

11. **struct nodeList** - a structure to store a list of nodes. It contains one member:

**nodes[5][15**] - a two-dimensional character array to store the IP addresses of up to five nodes.

12. **struct IPC** - a structure to send data between parent and child processes using pipes. It contains four members:

**type[1]** - a character array to represent the type of the request.

**data[1024]** - a character array to store the data being sent or received.

**nl** - a structure of type nodeList to store a list of nodes.

**childPID** - an integer to store the process ID of the child process.

13. **struct manifest** - a structure to store information about the chunks of a file. It contains one member:

**chunk\_IP[5][2][15]** - a three-dimensional character array to store the IP addresses and names of up to five chunks of a file.

All of these variables are used in various functions throughout the code to perform different operations.

**Functions Of Client.c:**

**getIP():**

The getIP() function is responsible for obtaining the IP address of the current machine. It does this by calling the ifconfig command, which is used to configure network interfaces. The output of ifconfig is then parsed to extract the IP address. This function returns the IP address as a string.

**communicate():**

The communicate() function initiates the communication between the client and server. It sends the client's IP address to the server using the sendIP() function. Then, it requests the manifest data from the server using the getManifestData() function. The manifest data is a list of files that the client needs to download. After receiving the manifest data, the function calls getChunkFiles() to download each file corresponding to the entries in the manifest data.

**sendIP():**

The sendIP() function is used to send the client's IP address to the server. It sends the IP address as a string over a socket connection.

**getValidNodes():**

The getValidNodes() function is responsible for requesting a list of valid nodes that the client can connect to from the server. It sends a request to the server and then waits for the response. When the response is received, it sends the list of valid nodes to the parent process using a pipe.

**getManifestData():**

The getManifestData() function requests the server's manifest data, which is a list of files that the client needs to download. It sends a request to the server and waits for the response. The response contains the manifest data, which is then parsed and stored in a struct manifest.

**getChunkFiles():**

The getChunkFiles() function downloads the files corresponding to each entry in the manifest data. It sends a request to the server for each file and waits for the response. When the response is received, the function saves the file to the client's file system.

**sendNodeListToParent():**

The sendNodeListToParent() function sends the list of valid nodes to the parent process using a pipe. The parent process can then use this list to determine which node to connect to.

**clientSegmentForChunk():**

The clientSegmentForChunk() function is used to download a specific chunk of data from a server specified by its IP address and the chunk's filename. It sends a request to the server and waits for the response. When the response is received, the function saves the chunk of data to the client's file system.

**Server.c Code:**

This code is a program written in C language for a peer-to-peer file sharing system.

The program includes several header files that provide the necessary functions and structures for working with sockets, files, pipes, and signals. The main function is divided into several subroutines that handle different aspects of the system's functionality, such as getting the server IP address, dividing a file into different chunks, handling communication with the client, and handling the client's requests.

**Server Variables:**

struct request: This structure is used to send and receive requests between the server and client. It has three members: type, data, and filename. type is a character that indicates the type of request being made, data is a character array that can be used to send additional data with the request, and filename is used to specify a filename associated with the request.

struct clientIP: This structure is used to store the IP address of a client along with an index.

struct nodeList: This structure is used to store a list of IP addresses for multiple clients.

struct IPC: This structure is used to send and receive data between processes using pipes. It has several members, including type, data, IP, and filename. type is a character that indicates the type of data being sent or received, data is a character array that can be used to send additional data, IP is used to specify an IP address associated with the data, and filename is used to specify a filename associated with the data.

**getIP()**: This function is used to get the IP address of the server.

**distributeFile():** This function is used to divide a file into different chunks.

**communicate():** This function is used to handle communication with a single client.

**handleClientRequest():** This function is used to handle requests from clients and fire functions accordingly.

**recvClientIP():** This function is used to receive the IP address of a client.

**getClientsIP():** This function is used to get the IP addresses of all connected clients.

**sendManifestData():** This function is used to send the manifest data to a client.

**sendChunkFile():** This function is used to send a chunk file to a client.

**updateManifest():** This function is used to update the manifest data on the server.