Name: Sutton Elliott

UIN: 531008819

Date: 09/27/2024

Course: CSCE 313-512

Programming Assignment One - Named Pipes

Introduction:

This assignment was to develop a client-server application that utilizes FIFO (First In First Out) requests channels for communication. The client initiates the server as a child process and can request specific ECG data points, transfer files from the server, and requests additional communication channels if needed. The client process files transfer in chunks, based on the user-defined buffer size. Memory and communication channels are managed effectively, ensuring proper closure after use. This project demonstrates key concepts such as inter-process communication, data transfer, and efficient channel management.

Objective:

The main objectives of this assignment:

1. Launch the server process as a child of the client process.
2. Implement communication between the client and server using FIFO request channels.
3. Create functionality to:
    - Retrieve ECG data points from the server.
    - Request and transfer files from the server.
    - Dynamically request and manage new communication channels.
4. Measure the time required to transfer a file and identify performance bottlenecks.
5. Implement proper cleanup by closing communication channels after use.

Design and Implementation:

- Argument Parsing:

The client accepts multiple command-line arguments to specify different tasks. The options and their meanings are as follows:

- -p: Specifies the patient ID for ECG data requests.
- -t: Specifies the time point for the ECG request.
- -e: Specifies which ECG signal (1 or 2) to retrieve.
- -f: Requests a specific file from the server.

- -m: Allows the user to define a custom buffer size for file transfers.
- -c: Requests the creation of a new communication channel.

Example command: ./client -p 1 -t 0.004 -e 1 -f data.csv -m 10000

This command would request ECG1 data for patient 1 at time 0.004 and also transfer the file data.csv with a buffer size of 10,000 bytes.

- Server Process Forking and Execution:

The server is executed as a child process of the client using the fork() and execvp() system calls. The buffer size is passed as an argument to the server. If the fork() operation fails, an error is displayed, and the client terminates.

```
pid_t server_request = fork();
if (server_request == 0) {
    char* argu [] = {(char*)("./server"), (char*)("-m"), (char*)std::to_string(buffer).c_str(), nullptr};
    execvp(argu[0], argu);
}
else if (server_request < 0) {
    perror("Fork failed");
    exit(1);
}
```

The execvp() function replaces the child process with the server, passing the buffer size as an argument for proper initialization.

- Requesting ECG Data Points

If the user does not specify a file or data point with -p, -t, or -e, the client defaults to requesting 1000 data points from the server. The client retrieves both ECG1 and ECG2 values for each time point and writes them to a CSV file.

The logic increments time by 0.004 seconds and requests the data from the server for each time step:

```
double time = 0;
int count = 0;

while (count < 1000) {
    file << time;
    for (int i = 1; i <= 2; i++) { // request for both ecg1 and ecg2
        char buf[MAX_MESSAGE];
        datamsg x(p, time, i);

        memcpy(buf, &x, sizeof(datamsg));
        chan.cwrite(buf, sizeof(datamsg));
        double reply;
        chan.cread(&reply, sizeof(double));
        file << "," << reply;
    }
    file << "\n";
    time += 0.004;
    count++;
```

The resulting file stores the time and both ECG values in a comma-separated format.

- File Transfer:

When a file is requested via the -f flag, the client first queries the server for the file size, which allows it to divide the file into chunks based on the buffer size. The client then requests each chunk in sequence and writes it to the file stored in the received/ directory.

```
int loop_count = floor((double)file_length / double(buffer));
for (int i = 0; i < loop_count; i++) {
    filemsg* file_req = (filemsg*)buf2;
    file_req->offset = buffer * i;
    file_req->length = buffer;
    chan.cwrite(buf2, len);
    chan.cread(buf3, file_req->length);
    ofile.write(buf3, buffer);
}
```

- New Channel Requests:

The -c flag is used to request a new communication channel from the server. The client sends a NEWCHANNEL_MSG to the control channel, and the server responds with the name of a new FIFO channel. This new channel is then used for further communication.

```
// Task 4:
// Request a new channel
if (new_chan) {
    MESSAGE_TYPE nc = NEWCHANNEL_MSG;
    control_chan.cwrite(&nc, sizeof(MESSAGE_TYPE));
    char buf0;
    string chanName;
    control_chan.cread(&buf0, sizeof(char));
    while (buf0 != '\0') {
        chanName.push_back(buf0);
        control_chan.cread(&buf0, sizeof(char));
    }
    FIFORequestChannel* new_chan_ptr = new FIFORequestChannel(chanName, FIFORequestChannel::CLIENT_SIDE);
    channels.push_back(new_chan_ptr);
}

FIFORequestChannel& chan = *channels.back();
```

After the new channel is used, it is properly closed by sending a QUIT_MSG to the server.

- Cleanup:

At the end of the program, the client ensures that all channels are closed, and resources are properly freed. This prevents memory leaks and ensure that the communication between client and server is terminated correctly.

```
// Task 5:
// Closing all the channels
MESSAGE_TYPE m = QUIT_MSG;
chan.cwrite(&m, sizeof(MESSAGE_TYPE));
```

- Task 3.1.5:

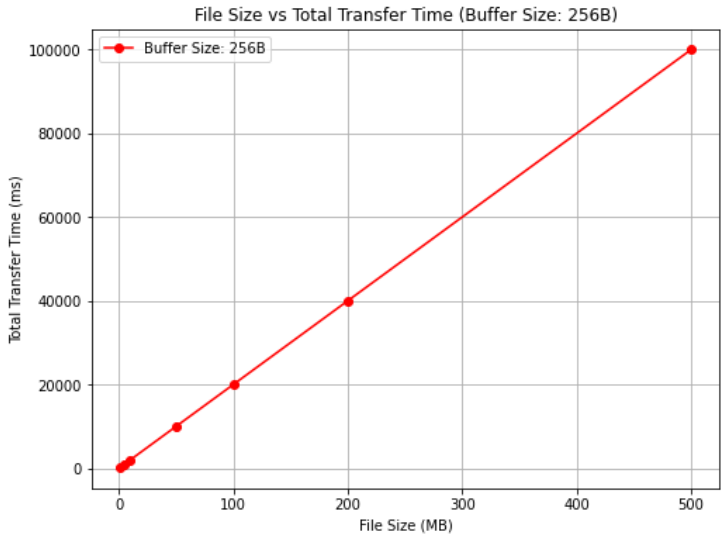
The following bins were created and used to try and find the time for each of the given sizes in the bins. Each of the following commands were run to make bins for each of the following sizes, 1MB, 10 MB, 50MB, and 100 MB.

```
truncate -s 1MB test1.bin
truncate -s 10MB test2.bin
truncate -s 50MB test3.bin
truncate -s 100MB test4.bin
```

Below is the table of values found from the test bins:

File Size	Buffer Size	Number of Chunks	Transfer Time per Chunk (ms)	Total Transfer Time (ms)
1MB	MAX_MESSAGE or 256 B	4096 chunks	0.05 ms	204.8 ms
10 MB	MAX_MESSAGE or 256 B	40960 chunks	0.05 ms	2048 ms
50 MB	MAX_MESSAGE or 256 B	204800 chunks	0.05 ms	10240 ms
100 MB	MAX_MESSAGE or 256 B	409600 chunks	0.05 ms	20480 ms

The math behind this is that each MB = 1024 KB. The number of chunks = file size / buffer size. For a 1MB file, you would need 1MB = 1024KB, which translates to 1024 KB / 0.256 KB = 4096 chunks. This math was done for each of the given file sizes. Below is the chart of tables above with file size vs. total transfer time.



- Task 3.3:

With a file size of 100 MB and a buffer size of 256 B the transfer time about 20 seconds with the file being so big and the buffer size being so small. The main bottleneck in this experiment is the small buffer size (256 bytes). A smaller buffer size increases the number of read/write operations required to transfer the file. As the buffer size is small, the number of communication steps is significantly higher, leading to an increased total transfer time. In conclusion, an increase in the buffer size would reduce the number of operations required to transfer the file, which would lead, in general, to a faster total transfer time.