Assignment 3

Question

Create two processes. Transfer 1GB file from process1 to process2 using a Socket. Now, transfer that file from process2 to process1 using the same Socket.Now, compare the two files to make sure that the same file has returned back. Also, print the time required to do this double transfer. Attach this output to the source file as a comment. Please note that, you can see the socket which your program creates. There are also various bash shell commands available to see the sockets created by the program. So, once your program creates the socket, make sure you use proper command to see the socket info and paste that output as a comment in your source file.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/un.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h> // Added for time measurement

#define CHUNK_SIZE 4096

int main(){
    int sockfd;
    // struct sockaddr_in serv_addr;
    struct sockaddr_un server_addr, client_addr;
    socklen_t client_addrlen = sizeof(client_addr);
    int newsockfd, listen_fd;

    char buffer[CHUNK_SIZE];
    ssize_t bytes_received;

    char* socket_path = "/tmp/socket_file";

    // Fork to create Process 2 (child process)
    pid_t child_pid = fork();

    if (child_pid == -1){
        perror("fork");
        return 1;
    }

    if (child_pid == 0){
        // This is Process 2 (child process)
        memset(&server_addr, 0, sizeof server_addr);
        server_addr.sun_family = AF_UNIX;
        strncpy(server_addr.sun_path, socket_path, sizeof(server_addr.sun_path) - 1);

        sleep(2); // wait for server to set up socket

        newsockfd = socket(AF_UNIX, SOCK_STREAM, 0);
        connect(newsockfd, (struct sockaddr *)&server_addr, sizeof server_addr);

        // Create a data structure (e.g., an array) to temporarily store chunks
        char data_store[CHUNK_SIZE];
        size_t data_store_size = 0;

        while ((bytes_received = read(newsockfd, (void *)buffer, CHUNK_SIZE)) > 0){
            if (write(newsockfd, (void *)buffer, bytes_received) < 0){
                perror("write to socket_recv failed in P2");
                exit(1);
            }
            if (bytes_received < CHUNK_SIZE)
                break;
        }
        // Close the socket and exit
        close(newsockfd);
    }
    else{
        // This is Process 1 (parent process)
        // Create a TCP socket
        sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sockfd == -1){
            perror("socket");
            return 1;
        }

        // Set up the server address structure
        memset(&server_addr, 0, sizeof server_addr);
        server_addr.sun_family = AF_UNIX;
        strncpy(server_addr.sun_path, socket_path, sizeof(server_addr.sun_path) - 1);

        if((listen_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0){
            perror("Listen failed");
            exit(1);
        }
        if((bind(listen_fd, (struct sockaddr *)&server_addr, sizeof server_addr)) < 0)
{
            perror("Bind failed");
            exit(1);
        }
        if(listen(listen_fd, 5) < 0){
            perror("Listening failed");
            exit(1);
        }
        newsockfd = accept(listen_fd, (struct sockaddr *)&client_addr,
&client_addrlen);
```

```c
        // Open the file to send (large_file.txt in /tmp)
        FILE *file = fopen("/tmp/large_file.txt", "rb");
        if (file == NULL){
            perror("fopen");
            return 1;
        }

        // Create and open the received file in /tmp
        FILE *received_file = fopen("/tmp/received_file.txt", "wb");
        if (received_file == NULL){
            perror("fopen received_file");
            return 1;
        }

        struct timeval start_time, end_time; // Added for time measurement

        gettimeofday(&start_time, NULL); // Start timing

        if(system("ss -x -a") < 0){ // For socket information
            perror("Internal error in command");
            exit(1);
        }

        // Send the file in chunks
        while ((bytes_received = fread(buffer, 1, CHUNK_SIZE, file)) > 0){
            if ((send(newsockfd, buffer, bytes_received, 0)) < 0){
                perror("Write to socket_path failed in P1");
                exit(1);
            }
            /* continuously read from the receive FIFO and keep writing to the new
large file */
            if ((bytes_received = recv(newsockfd, buffer, CHUNK_SIZE, 0)) > 0){
                if (fwrite(buffer, 1, bytes_received, received_file) < 0){
                    perror("write to lfile_recv failed in P1");
                    exit(1);
                }
            }
        }

        gettimeofday(&end_time, NULL); // End timing

        long long elapsed_time = (end_time.tv_sec - start_time.tv_sec) * 1000000LL +
(end_time.tv_usec - start_time.tv_usec);
        printf("Transfer Time: %lld microseconds\n", elapsed_time);

        if(fclose(file) < 0 || fclose(received_file) < 0 || close(newsockfd) < 0){ //
Closing the files
            perror("Problem occured in closing the files");
            exit(1);
        }

        // Compare the sent and received files
        int compare_result = system("diff -q /tmp/large_file.txt
/tmp/received_file.txt");
        if (compare_result == 0){
            printf("Sent and received files are the same!\n");
        }
        else{
            printf("Sent and received files are different.\n");
        }
        if(unlink(socket_path) < 0){ // Unlinking the socket file
            perror("Socket file could not be removed");
            exit(1);
        }
        // if(system("rm /tmp/socket_file") < 0){
        //     perror("Cannot remove socket file");
        //     exit(1);
        // }
    }
    return 0;
}
```

Explaination

This C program demonstrates interprocess communication between two processes, Process 1 and Process 2, using Unix domain sockets. It involves transferring a large file from Process 1 to Process 2, measuring transfer time, and checking the integrity of the received file. Let's break down the code and explain it step by step:

1. Header Files:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/un.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>
```

- These header files provide the necessary functions and data structures for socket programming, file operations, and time measurement.

2. Constants:

```
#define CHUNK_SIZE 4096
```

- `CHUNK_SIZE` defines the size of data chunks used during file transfer. In this code, it's set to 4096 bytes.

3. Socket Paths:

```
char* socket_path = "/tmp/socket_file";
```

- `socket_path` is a variable in the code that defines the path to the Unix domain socket file used for interprocess communication between the two processes, Process 1 and Process 2. In this code, it is set to `"/tmp/socket_file"`.

Here's what it represents:

- **`socket_path`**: This variable is a string that specifies the path to the Unix domain socket file. Unix domain sockets are used for local interprocess communication on the same machine. The path specifies where the socket file will be created and how processes can connect to it.

- In the code, the Unix domain socket is created with the path `"/tmp/socket_file"`.

  - This means that both Process 1 and Process 2 will use this socket file for communication.
  - Process 1 will bind and listen to this socket, and Process 2 will connect to it.

- It's important to note that the specified path should be accessible to both processes, and the file shouldn't already exist before running the code to avoid conflicts.

  - If the file already exists, it may need to be removed or specified with a unique name.

- `socket_path` is a variable that represents the path to a Unix domain socket file.

  - This Unix domain socket file is used for interprocess communication (IPC) between two processes (Process 1 and Process 2) running on the same machine.
  - The socket file itself is not a regular file where data is written; instead, it serves as a communication endpoint.

4. Forking a Child Process (Process 2):

```c
pid_t child_pid = fork();

if (child_pid == -1) {
   perror("fork");
   return 1;
}
```

- The program forks a child process to create Process 2. The parent process (Process 1) and the child process (Process 2) will communicate through the Unix domain socket.
- In the provided code, Process 1 (the parent process) and Process 2 (the child process) are created using the `fork()` system call.
- The `fork()` call essentially creates a new process that is a copy of the current process, and both processes continue execution from the point where `fork()` was called.

Sequence of events between Process 1 and Process 2:

1. The `fork()` call is made, creating a new process (Process 2) that is a copy of the parent process (Process 1). At this point, both processes have the same code and variables.
2. After the `fork()` call, both Process 1 and Process 2 exist simultaneously and execute the same code. However, they can distinguish themselves based on the return value of `fork()`. In Process 1, `fork()` returns the child's process ID (PID), while in Process 2(Child Process), it returns 0.
3. Process 1 and Process 2 continue to execute the code sequentially, starting from the point immediately after the `fork()` call.
4. In our code, Process 1 and Process 2 have different roles. Process 1 is responsible for setting up the server, binding to a Unix domain socket, and sending data. Process 2 is responsible for connecting to the server and receiving data.
5. The logic inside the `if` and `else` blocks helps distinguish the roles of Process 1 and Process 2 based on the return value of `fork()`.

Here's a simplified breakdown:

- Process 1 (parent):
    - It performs the actions within the `if` block because `fork()` returned the child's PID (which is greater than 0).
    - This includes setting up the server, binding to the socket, and sending data.
    - After completing its tasks in the `if` block, it may wait for Process 2 to complete its tasks.
- Process 2 (child):
    - It performs the actions within the `else` block because `fork()` returned 0.
    - This includes connecting to the server (which was set up by Process 1) and receiving data.
    - After completing its tasks in the `else` block, it may exit.

So, Process 2 doesn't need to wait explicitly for Process 1 to set up the server.

- Both processes proceed based on their respective roles, and the code execution continues sequentially for each process after the `fork()` call.
    - Process 1 sets up the server, and Process 2 connects to it when it's its turn to execute the `else` block.
    - The key is that both processes are executing concurrently and can perform their tasks independently based on the logic in the code.

5. Process 1 (Parent Process, Server Side) Implementation:

```
else {
   // This is Process 1 (parent process)
   // Implementation of Process 1
}
```

Process 1 is the parent process responsible for sending a large file to Process 2 using a Unix domain socket. Here's a step-by-step breakdown of how Process 1 works:

1. **Socket Initialization:**

   ```c
   sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
   if (sockfd == -1) {
       perror("socket");
       return 1;
   }
   ```

   - Process 1 begins by creating a Unix domain socket using the `socket()` function. It specifies the socket family as `AF_UNIX` (indicating a Unix domain socket) and the socket type as `SOCK_STREAM` (indicating a stream socket for reliable, bidirectional communication).

2. **Server Address Setup:**

   ```c
   memset(&server_addr, 0, sizeof server_addr);
   server_addr.sun_family = AF_UNIX;
   strncpy(server_addr.sun_path, socket_path, sizeof(server_addr.sun_path) - 1);
   ```

   - Process 1 sets up the server address structure, which includes the socket family (`AF_UNIX`) and the path to the Unix domain socket file specified by `socket_path` .

3. **Socket Binding and Listening:**

   ```c
   if ((listen_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0){
       perror("Listen failed");
       exit(1);
   }
   if ((bind(listen_fd, (struct sockaddr *)&server_addr, sizeof server_addr)) < 0)
   {
       perror("Bind failed");
       exit(1);
   }
   if (listen(listen_fd, 5) < 0){
       perror("Listening failed");
       exit(1);
   }
   ```

   - Process 1 creates a new socket, `listen_fd` , for listening to incoming connections. This socket is used solely for accepting connections from Process 2.
   - It binds the socket to the Unix domain socket file specified in `server_addr` (`socket_path` in this case).
   - Process 1 listens for incoming connections with a backlog queue size of 5 (indicating that up to 5 connection requests can be queued).

4. **Accepting Connections:**

   ```c
   newsockfd = accept(listen_fd, (struct sockaddr *)&client_addr,
   &client_addrlen);
   ```

    - Once Process 2 is ready to connect, Process 1 accepts the connection using the `accept()` function.
    - It's important to note that `accept()` is a blocking function. It will block the execution of your program until a client connects. If no clients are connecting, the program will wait indefinitely. This is typically the desired behavior for server programs as they need to wait for incoming connections.
    - `newsockfd` is a new socket descriptor created for the established connection. It represents the communication channel with Process 2.

5. **Opening the Source File:**

   ```
   FILE *file = fopen("/tmp/large_file.txt", "rb");
   if (file == NULL){
       perror("fopen");
       return 1;
   }
   ```

    - Process 1 opens the source file, which is named "large_file.txt" and located in the "/tmp" directory, in binary read mode (`"rb"`).

6. **Opening the Received File:**

   ```
   FILE *received_file = fopen("/tmp/received_file.txt", "wb");
   if (received_file == NULL){
       perror("fopen received_file");
       return 1;
   }
   ```

    - Process 1 opens an output file, "received_file.txt," also located in the "/tmp" directory, in binary write mode (`"wb"`). This is where the received data will be written.

7. **Transfer Loop - Sending Data:**

   ```
   while ((bytes_received = fread(buffer, 1, CHUNK_SIZE, file)) > 0){
       if ((send(newsockfd, buffer, bytes_received, 0)) < 0){
           perror("Write to socket_path failed in P1");
           exit(1);
       }
   }
   ```

    - Process 1 enters a loop to read data from the source file in chunks of `CHUNK_SIZE` bytes (4,096 bytes in this case) using `fread()` .
    - It sends each chunk of data to Process 2 through the Unix domain socket using the `send()` function.

8. **Transfer Loop - Receiving Data:**

   ```
   if ((bytes_received = recv(newsockfd, buffer, CHUNK_SIZE, 0)) > 0){
       if (fwrite(buffer, 1, bytes_received, received_file) < 0){
           perror("write to lfile_recv failed in P1");
           exit(1);
       }
   }
   ```

- Simultaneously, Process 1 receives data from Process 2 through the Unix domain socket using the `recv()` function.
- It writes the received data to the "received_file.txt" file using `fwrite()`.

9. **Transfer Time Measurement:**

```
gettimeofday(&end_time, NULL); // End timing
long long elapsed_time = (end_time.tv_sec - start_time.tv_sec) * 1000000LL +
(end_time.tv_usec - start_time.tv_usec);
printf("Transfer Time: %lld microseconds\n", elapsed_time);
```

- Process 1 measures the transfer time by recording the start and end times using the `gettimeofday()` function.
- It calculates the elapsed time in microseconds and prints it to the console.

10. **File and Socket Cleanup:**

```
if (fclose(file) < 0 || fclose(received_file) < 0 || close(newsockfd) < 0){
   perror("Problem occurred in closing the files");
   exit(1);
}
```

- Process 1 closes the source file, received file, and the socket descriptor to release system resources.

11. **File Comparison:**

```
int compare_result = system("diff -q /tmp/large_file.txt
/tmp/received_file.txt");
if (compare_result == 0){
   printf("Sent and received files are the same!\n");
}
else{
   printf("Sent and received files are different.\n");
}
```

- It uses the `system()` function to run the `diff` command and compare the original source file ("/tmp/large_file.txt") with the received file ("/tmp/received_file.txt").
- Depending on the result, it prints a message indicating whether the files are the same or different.

12. **Socket File Cleanup:**

```
if (unlink(socket_path) < 0){
   perror("Socket file could not be removed");
   exit(1);
}
```

- Finally, Process 1 unlinks (removes) the Unix domain socket file specified by `socket_path` to clean up the socket file after the communication is complete.

6. Process 2 (Child Process, Client Side) Implementation:

```
// Child Process Creation
pid_t child_pid = fork();
if (child_pid == -1){
    perror("fork");
    return 1;
}
if (child_pid == 0){
    // This is Process 2 (child process)
```

1. **Child Process Creation**:
   - Process 2 is created as a child process using `fork()` . It checks the return value of `fork()` to distinguish itself as the child process (where `child_pid` is 0).

```
// Initialization
memset(&server_addr, 0, sizeof server_addr);
server_addr.sun_family = AF_UNIX;
strncpy(server_addr.sun_path, socket_path, sizeof(server_addr.sun_path) - 1);

sleep(2); // Wait for server to set up socket
```

2. **Initialization**:
   - Process 2 initializes the `server_addr` structure with the address information it needs to connect to the Unix domain socket server.
   - It sets `sun_family` to `AF_UNIX` to specify that it's using Unix domain sockets.
   - The `sun_path` field of `server_addr` is set to the same path as `socket_path` , ensuring that it connects to the server created by Process 1.
   - The `sleep(2)` function introduces a brief delay to allow Process 1 to set up the server. This ensures that Process 2 doesn't attempt to connect before the server is ready.

```
// Socket Creation and Connection
newsockfd = socket(AF_UNIX, SOCK_STREAM, 0);
connect(newsockfd, (struct sockaddr *)&server_addr, sizeof server_addr);
```

3. **Socket Creation and Connection**:
   - Process 2 creates a new Unix domain socket using `socket()` . The socket descriptor is stored in `newsockfd` .
   - It then uses `connect()` to establish a connection to the Unix domain socket server. The server's address is specified in the `server_addr` structure(which is `socket_path` in this case).

```
// Data Reception
char data_store[CHUNK_SIZE];
size_t data_store_size = 0;

while ((bytes_received = read(newsockfd, (void *)buffer, CHUNK_SIZE)) > 0){
    if (write(newsockfd, (void *)buffer, bytes_received) < 0){
        perror("write to socket_recv failed in P2");
        exit(1);
    }
    if (bytes_received < CHUNK_SIZE)
        break;
}
```

4. **Data Reception**:
   - Process 2 enters a loop where it reads data from the socket using `read()`.
   - The received data is stored in the `buffer` array, and the loop continues as long as there is more data to read.
   - In this code, Process 2 echoes the received data back to the server by using `write()` on the same socket. This step is optional and can be modified to perform different operations with the data.

```
// Loop Continuation and Exit
close(newsockfd);
```

5. **Loop Continuation and Exit**:
   - After processing the data and ensuring that there's no more data to read, Process 2 closes the socket (`newsockfd`) to release resources.
   - Process 2 can then exit, ending its execution.

In summary, Process 2's role is to connect to the Unix domain socket server created by Process 1, receive data from it, and optionally perform operations with the received data. The code is structured to ensure a proper connection and data exchange between the two processes, allowing them to communicate securely and efficiently on the same machine.

Why UNIX Sockets?

Unix domain sockets, often referred to simply as Unix sockets, are a type of interprocess communication (IPC) mechanism used for communication between processes running on the same Unix-like operating system, such as Linux or macOS. Unlike network sockets that communicate over a network, Unix domain sockets operate entirely within the operating system's kernel and do not involve physical network hardware. They offer several advantages for local communication:

1. **Efficiency:** Unix domain sockets are implemented entirely within the operating system's kernel, making them highly efficient. Data transfer between processes occurs in-memory, without the overhead associated with network protocols.

2. **Low Latency:** Since Unix domain sockets do not involve the network stack, they have lower latency compared to network sockets, making them suitable for high-performance local communication.

3. **Security:** Unix domain sockets are protected by file permissions, allowing fine-grained control over who can access and communicate through them. This provides a level of security and isolation.

4. **Stream and Datagram Sockets:** Unix domain sockets support both stream-oriented (SOCK_STREAM) and datagram-oriented (SOCK_DGRAM) communication, offering

flexibility for different types of IPC.

5. **Named Sockets:** Unix domain sockets can be named using a file path within the file system. This allows processes to communicate by specifying a well-known path, making it easier to locate and connect to sockets.

6. **Bi-Directional Communication:** Unix domain sockets support bidirectional communication, allowing both processes to send and receive data.

Common use cases for Unix domain sockets include:

- **Interprocess Communication (IPC):** Processes running on the same machine can use Unix domain sockets to exchange data or control messages.

- **Server-Client Communication:** A server process can create a Unix domain socket and listen for incoming connections, while client processes connect to the server to request services or exchange data.

- **Local Communication:** Services that need to communicate with each other locally, such as database servers, can use Unix domain sockets for efficient and secure communication.

- **System Services:** Many Unix-like operating systems use Unix domain sockets for system services like X Window System (X11) communication, D-Bus for interprocess communication, and more.