CSN-361

Assignment -2

Abhijeet Shakya
17114002

Problem Statement 1

Write a socket program in C to connect two nodes on a network to communicate with each other, where one socket listens on a particular port at an IP, while other socket reaches out to the other to form a connection.

SOLUTION:

The solution has 2 programs:

Server : This program creates a node which has a socket to listen on the port 8080 (localhost).

`int sockfd = socket(AF_INET, SOCK_STREAM,0)` creates a socket with IPv4 communication domain and TCP communication type.

`setsockopt()` method helps in manipulating options for the socket referred by the file descriptor sockfd.

`bind()` method binds the socket to the address and port number specified in addr(custom data structure).

`listen()` method puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection.

`accept()` method extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket.

`sockaddr_in` is a struct for all syscalls and functions that deal with internet addresses.

```
#include <unistd.h>

#include <stdio.h>

#include <sys/socket.h>

#include <stdlib.h>

#include <netinet/in.h>

#include <string.h>



#define PORT 8080



int main(int argc, char const *argv[]) {



    int sockfd, newSocket, valRead;

    struct sockaddr_in address;

    int opt = 1;

    int addrlen = sizeof(address);
```

```
    char buffer[1024] = {0};
    char *response = "Server responded!";    // Response message from server

    sockfd = socket(AF_INET, SOCK_STREAM, 0);    // File descripter for socket with IPv4 and TCP.

    if(sockfd == 0) {
            perror("socket failed!");
            exit(EXIT_FAILURE);
    }

    // Attach the socket to port 8080 forcefully
    if(setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
            perror("setsockopt");
            exit(EXIT_FAILURE);
    }
```

```
   address.sin_family = AF_INET;

   address.sin_addr.s_addr = INADDR_ANY;     // For localhost

          address.sin_port = htons(PORT);



          // Bind the socket to localhost 8080

          if (bind(sockfd, (struct sockaddr *)&address, sizeof(address))<0) {

          perror("bind failed");

          exit(EXIT_FAILURE);

          }



          printf("Server listening...\n");



          if (listen(sockfd, 3) < 0) {     // Wait for client to make approach. Backlog here is 3 (max size of wait queue).

          perror("listen");
```

```
        exit(EXIT_FAILURE);

        }



        // Extract the first connection from pending queue and establish connection b/w client and server by creating a new socket.

        if ((newSocket = accept(sockfd, (struct sockaddr *)&address, (socklen_t*)&addrlen))<0) {

        perror("accept");

        exit(EXIT_FAILURE);

        }



        valRead = read( newSocket , buffer, 1024);

        printf("%s\n",buffer );

        send(newSocket , response , strlen(response) , 0 );

        printf("Response message sent\n");

        return 0;

}
```

Client : This program created a node which sends a request on port 8080 to be read by the server program.

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

#define PORT 8080

int main(int argc, char const *argv[])
{
        int sock = 0, valread;
        struct sockaddr_in serv_addr;
        char *request = "Client requesting...";
        char buffer[1024] = {0};

        if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
```

```
{
printf("\n Socket creation error \n");
return -1;
}

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);

// Convert IPv4 and IPv6 addresses from text to binary form
if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
{
printf("\nInvalid address/ Address not supported \n");
return -1;
}

if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
{
printf("\nConnection Failed \n");
return -1;
}

send(sock , request , strlen(request) , 0 );
printf("Request message sent\n");
valread = read( sock , buffer, 1024);
```

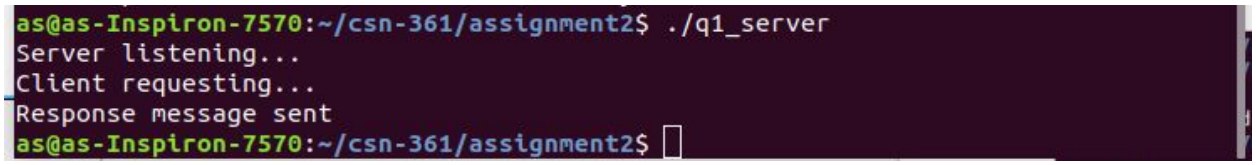
```
        printf("%s\n",buffer );
        return 0;
}
```

EXECUTION:

First we compile and run the server program. This sets up the server and the node starts listening for any calls.

Then the client program is compiled and run on a new terminal. As soon as the program is executed, it sends a message to the server and gets a response from the server.

The server program responds to the client when the request is made.



Problem Statement 2

Write a C program to demonstrate both Zombie and Orphan process.

SOLUTION:

The following program uses the fork() and sleep() system calls to create a simulation of orphan and zombie processes.

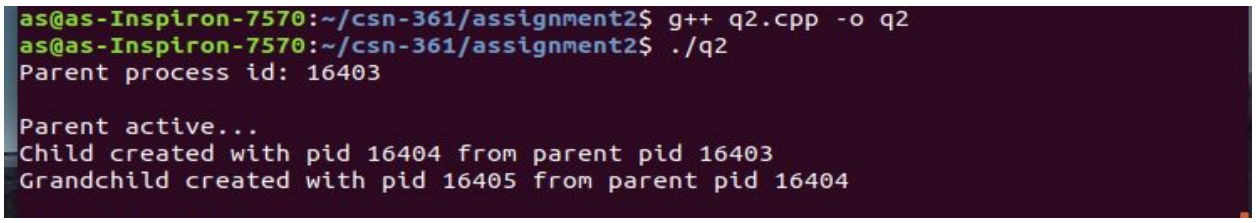
```
#include <bits/stdc++.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

using namespace std;

int main() {

    cout << "Parent process id: " << getpid() << endl << endl;
    pid_t child_pid = fork();

            if (child_pid > 0) {
```

```
        cout << "Parent active..." << endl;
        sleep(4);
        cout << "Parent terminated" << endl;
    }
    else if (child_pid == 0) {
        cout << "Child created with pid "<< getpid() << " from parent pid " << getppid() << endl;
        child_pid = fork();
        if(child_pid > 0) {
            sleep(1);
            cout << "Child sleeping..." << endl;
            sleep(2);
            cout << "Child awake again and active!" << endl;
            sleep(2);
            cout << "Child is now orphan!" << endl << endl;
        }
        else if(child_pid == 0) {
            cout << "Grandchild created with pid "<< getpid() << " from parent pid " <<
getppid() << endl << endl;
            sleep(1);
            cout << "Terminating grandchild" << endl;
            cout << "Grandchild is now zombie" << endl << endl;
        }
    }
```

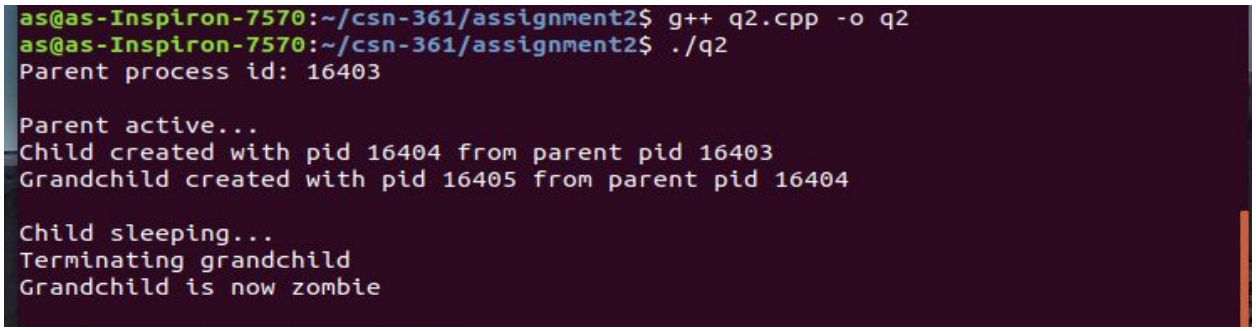
```
    return 0;
}
```

EXECUTION:

When the program is executed, a parent, a child and a grandchild are created.



After a few moments, the child process is put to sleep and while it is in the sleep state, the grandchild terminates. Now there is no grandchild process, but it exists in the table of the child process as its child.



In another few moments, the child process' state is reverted to active.

```
as@as-Inspiron-7570:~/csn-361/assignment2$ g++ q2.cpp -o q2
as@as-Inspiron-7570:~/csn-361/assignment2$ ./q2
Parent process id: 16403

Parent active...
Child created with pid 16404 from parent pid 16403
Grandchild created with pid 16405 from parent pid 16404

Child sleeping...
Terminating grandchild
Grandchild is now zombie

Child awake again and active!
```

In some time, the parent process terminates, thereby bringing up the terminal again to accept commands, but the child process is still running as an orphan.

The child process now runs as an orphan.

```
as@as-Inspiron-7570:~/csn-361/assignment2$ g++ q2.cpp -o q2
as@as-Inspiron-7570:~/csn-361/assignment2$ ./q2
Parent process id: 16403

Parent active...
Child created with pid 16404 from parent pid 16403
Grandchild created with pid 16405 from parent pid 16404

Child sleeping...
Terminating grandchild
Grandchild is now zombie

Child awake again and active!
Parent terminated
as@as-Inspiron-7570:~/csn-361/assignment2$ Child is now orphan!
```