Practice code 6

Problem Statement: Socket Programming in C

Design and implement a reliable and efficient network communication system using socket programming in C to enable data exchange between two or more processes running on different machines over a network.

Specific Requirements:

Socket creation: Create appropriate socket descriptors for the desired communication protocol (TCP, UDP, etc.).

Address binding: Bind the created socket to a specific network address and port number for both client and server applications.

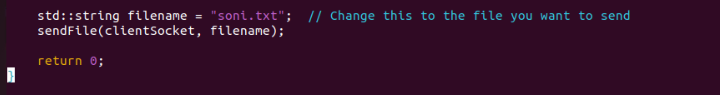
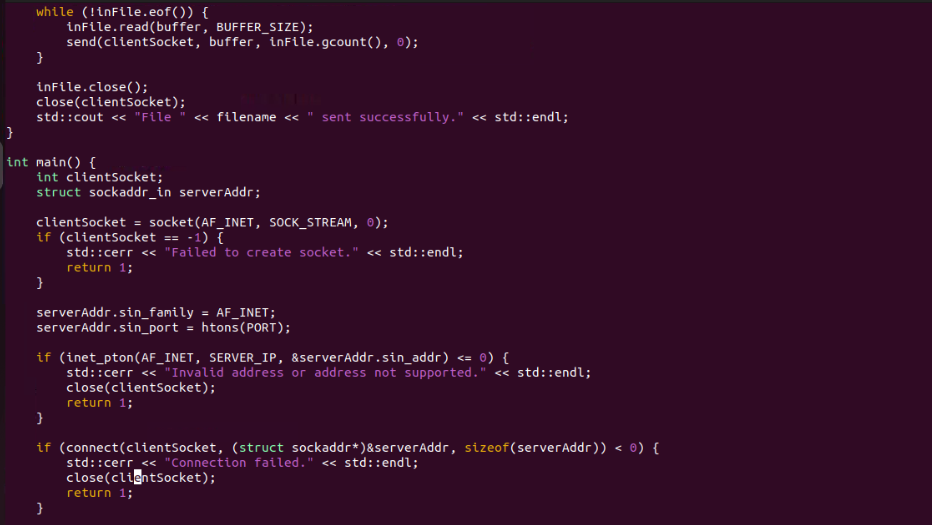
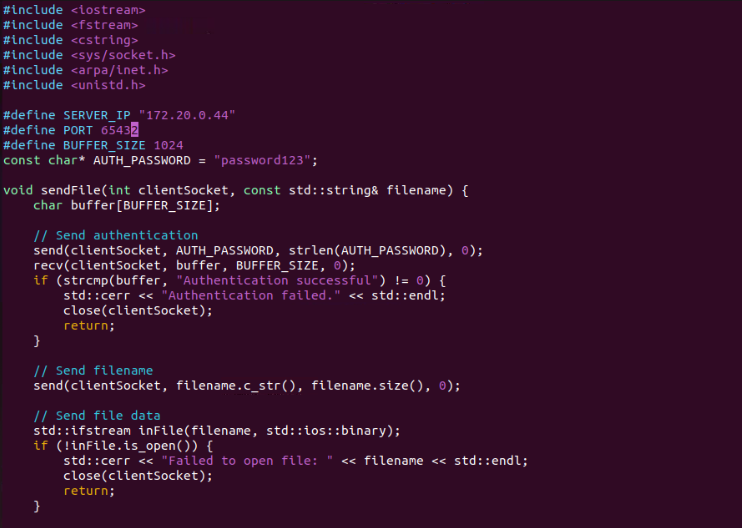
Connection establishment: Implement connection setup mechanisms (connect, accept) for TCP-based communication.

Data transfer: Develop functions for sending and receiving data over the established socket connection.

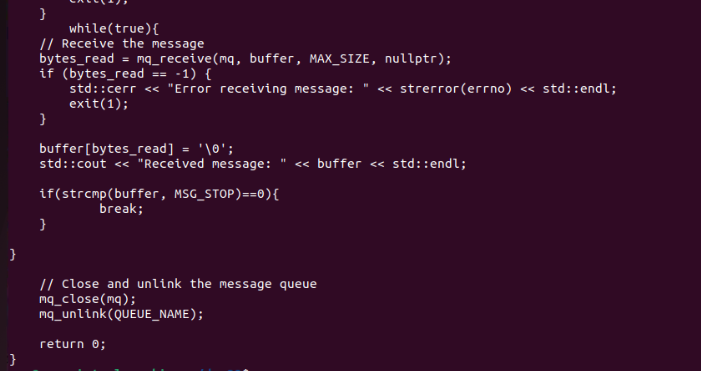
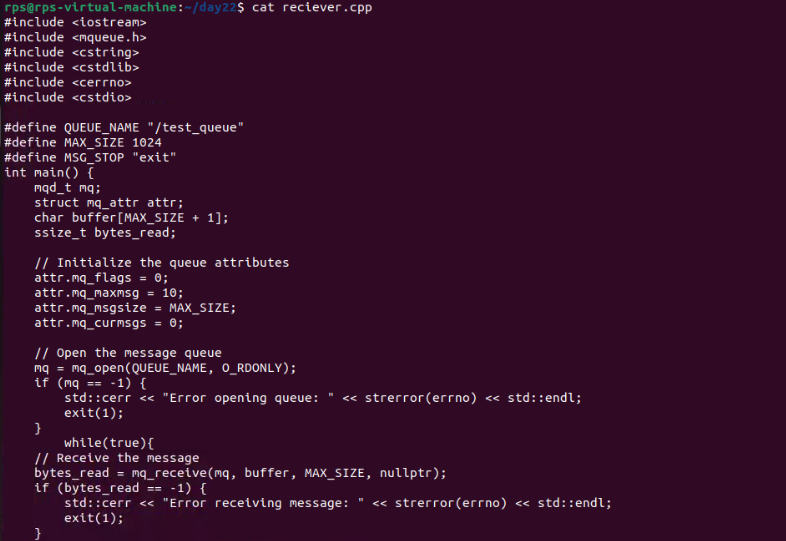
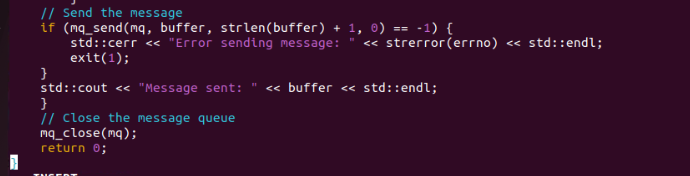
Error handling: Incorporate robust error handling mechanisms to address potential network issues and unexpected exceptions.

Concurrency: For server-side applications, consider handling multiple client connections concurrently using appropriate techniques (e.g., threading, forking).

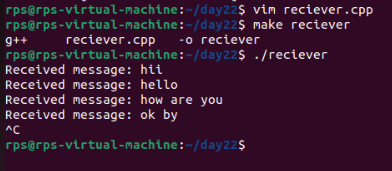
Security: Implement appropriate security measures to protect data integrity and confidentiality (e.g., encryption, authentication).



Sender and receiver code :



Output :



Process 1 and process 2

#include <iostream>

#include <fcntl.h>

#include <sys/mman.h>

#include <unistd.h>

#include <cstring>

#include "common.h"

int main() {

// Open shared memory

int shm\_fd = shm\_open(SHARED\_MEMORY\_NAME, O\_CREAT | O\_RDWR, 0666);

if (shm\_fd == -1) {

std::cerr << "Error opening shared memory: " << strerror(errno) << std::endl;

exit(1);

}

// Set size of shared memory

if (ftruncate(shm\_fd, sizeof(SharedMemory)) == -1) {

std::cerr << "Error setting size of shared memory: " << strerror(errno) << std::endl;

exit(1);

}

// Map shared memory

SharedMemory \*shared\_memory = (SharedMemory \*)mmap(nullptr, sizeof(SharedMemory), PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm\_fd, 0);

if (shared\_memory == MAP\_FAILED) {

std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;

exit(1);

}

// Initialize semaphores

sem\_t \*sem1 = sem\_open(SEMAPHORE1\_NAME, O\_CREAT, 0666, 1);

sem\_t \*sem2 = sem\_open(SEMAPHORE2\_NAME, O\_CREAT, 0666, 0);

if (sem1 == SEM\_FAILED || sem2 == SEM\_FAILED) {

std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;

exit(1);

}

// Initialize shared memory

shared\_memory->process1\_turn = true;

while (true) {

// Wait for process1\_turn to be true

sem\_wait(sem1);

if (!shared\_memory->process1\_turn) {

sem\_post(sem1);

continue;

}

// Write message to shared memory

std::cout << "Process1, enter a message: ";

std::cin.getline(shared\_memory->buffer, MAX\_SIZE);

shared\_memory->process1\_turn = false;

// Signal process 2

sem\_post(sem2);

if (strcmp(shared\_memory->buffer, "exit") == 0) break;

}

// Clean up

munmap(shared\_memory, sizeof(SharedMemory));

close(shm\_fd);

sem\_close(sem1);

sem\_close(sem2);

sem\_unlink(SEMAPHORE1\_NAME);

sem\_unlink(SEMAPHORE2\_NAME);

shm\_unlink(SHARED\_MEMORY\_NAME);

return 0;

}

Process2 :

#include <iostream>

#include <fcntl.h>

#include <sys/mman.h>

#include <unistd.h>

#include <cstring>

#include "common.h"

int main() {

// Open shared memory

int shm\_fd = shm\_open(SHARED\_MEMORY\_NAME, O\_CREAT | O\_RDWR, 0666);

if (shm\_fd == -1) {

std::cerr << "Error opening shared memory: " << strerror(errno) << std::endl;

exit(1);

}

// Set size of shared memory

if (ftruncate(shm\_fd, sizeof(SharedMemory)) == -1) {

std::cerr << "Error setting size of shared memory: " << strerror(errno) << std::endl;

exit(1);

}

// Map shared memory

SharedMemory \*shared\_memory = (SharedMemory \*)mmap(nullptr, sizeof(SharedMemory), PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm\_fd, 0);

if (shared\_memory == MAP\_FAILED) {

std::cerr << "Error mapping shared memory: " << strerror(errno) << std::endl;

exit(1);

}

// Initialize semaphores

sem\_t \*sem1 = sem\_open(SEMAPHORE1\_NAME, O\_CREAT, 0666, 1);

sem\_t \*sem2 = sem\_open(SEMAPHORE2\_NAME, O\_CREAT, 0666, 0);

if (sem1 == SEM\_FAILED || sem2 == SEM\_FAILED) {

std::cerr << "Error opening semaphores: " << strerror(errno) << std::endl;

exit(1);

}

while (true) {

// Wait for process1\_turn to be false

sem\_wait(sem2);

if (shared\_memory->process1\_turn) {

sem\_post(sem2);

continue;

}

// Read message from shared memory

std::cout << "Process2 received: " << shared\_memory->buffer << std::endl;

if (strcmp(shared\_memory->buffer, "exit") == 0) break;

// Write response to shared memory

std::cout << "Process2, enter a message: ";

std::cin.getline(shared\_memory->buffer, MAX\_SIZE);

shared\_memory->process1\_turn = true;

// Signal process 1

sem\_post(sem1);

if (strcmp(shared\_memory->buffer, "exit") == 0) break;

}

// Clean up

munmap(shared\_memory, sizeof(SharedMemory));

close(shm\_fd);

sem\_close(sem1);

sem\_close(sem2);

sem\_unlink(SEMAPHORE1\_NAME);

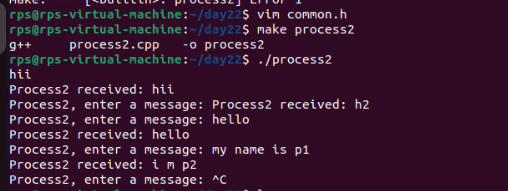
sem\_unlink(SEMAPHORE2\_NAME);

shm\_unlink(SHARED\_MEMORY\_NAME);

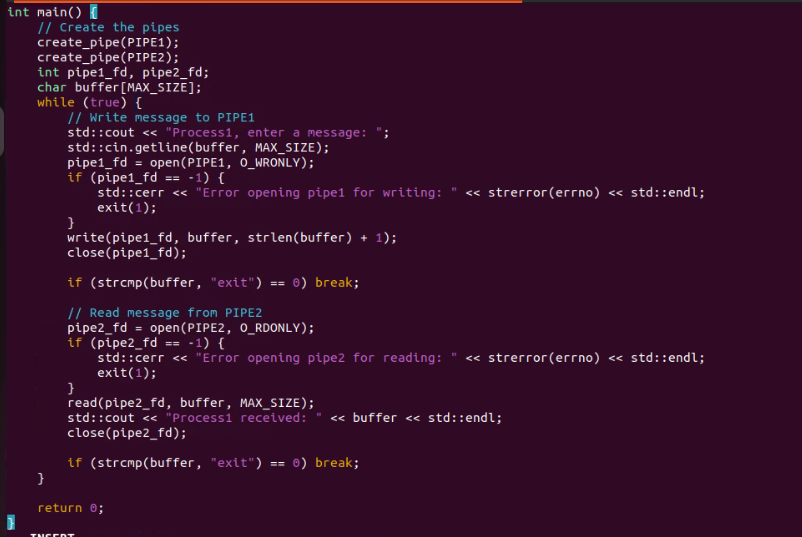
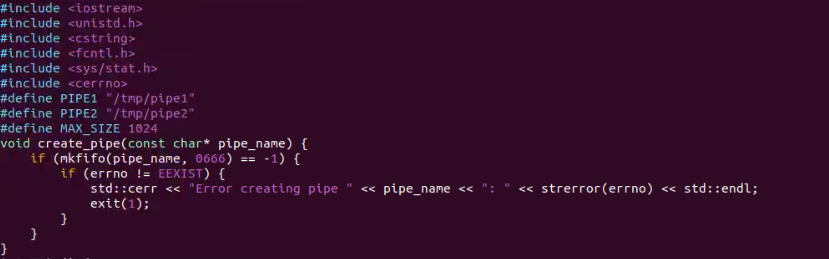
return 0;

}

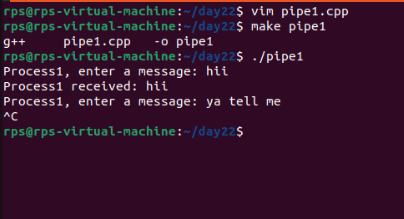
Output :



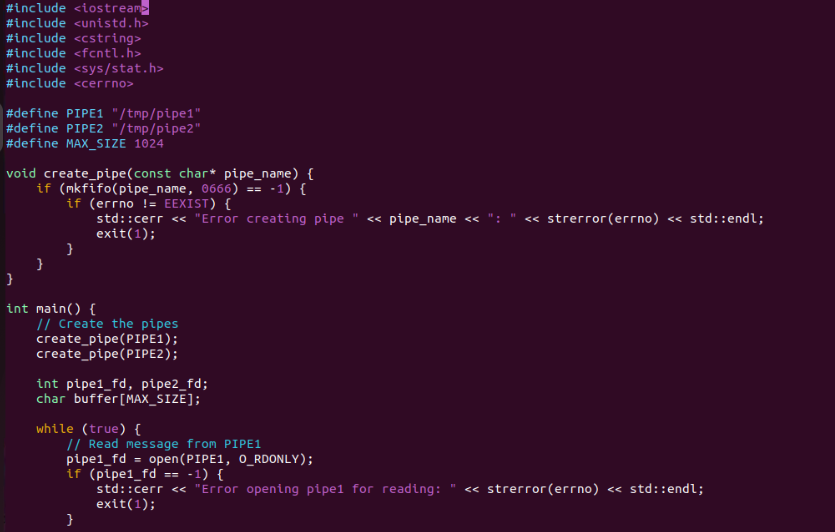
Pipe 1:

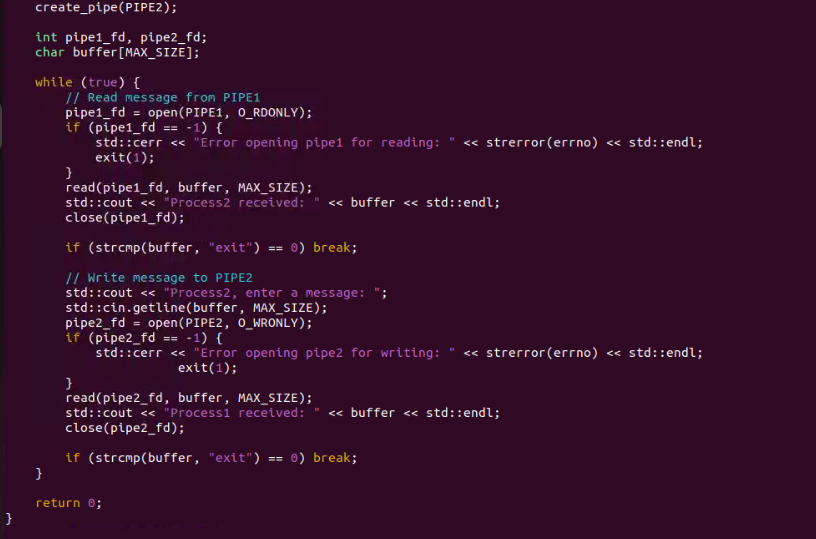


Output :

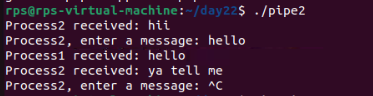


Pipe2 :



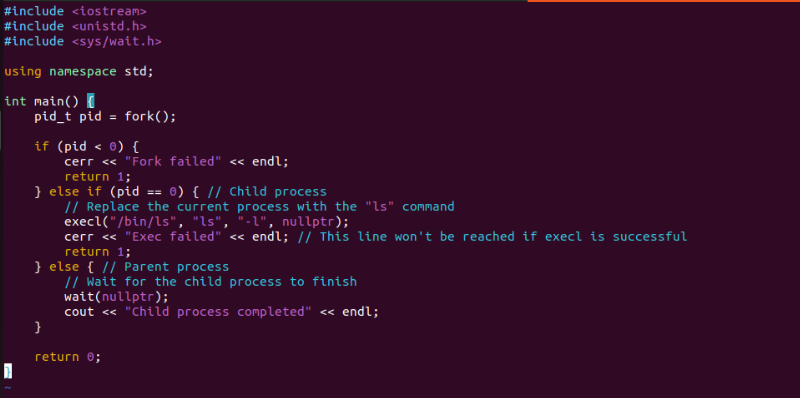


Output

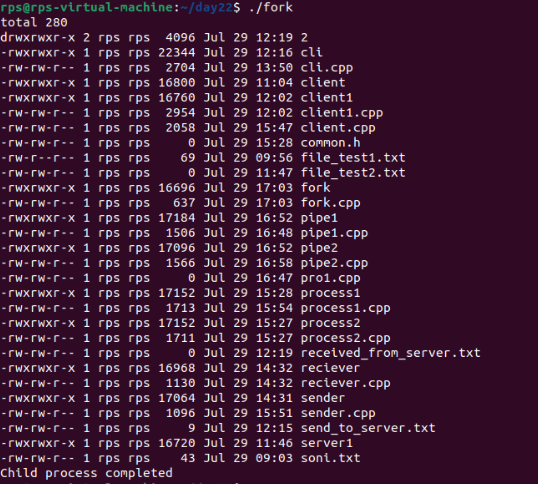


Fork is create a new child process

Code :

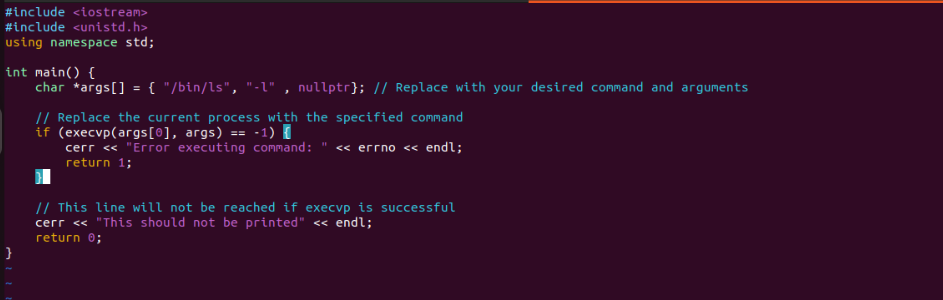


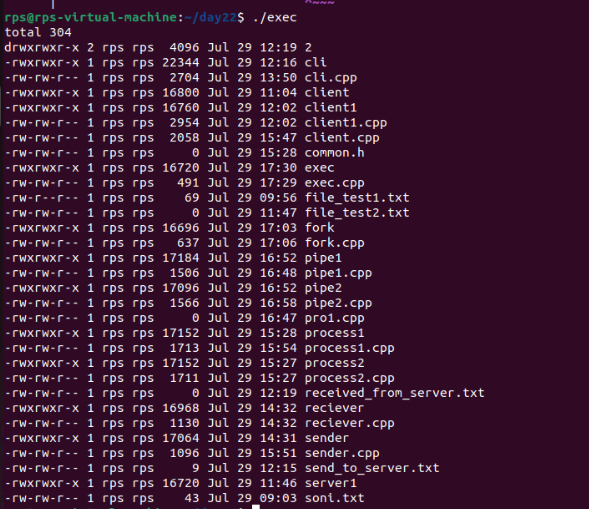
Output :



Exec is create a new process

Code :





Merge fork and exec code:

