Problem Statement: Inter-Process Communication (IPC) using Pipes, Shared Memory, and Message Queues

Design and implement efficient and reliable inter-process communication (IPC) mechanisms using pipes, shared memory, and message queues in C to facilitate data exchange and synchronization between multiple processes within a single system.

Specific Requirements:

Pipe: Create and manage unidirectional and bidirectional pipes for simple data transfer between related processes.

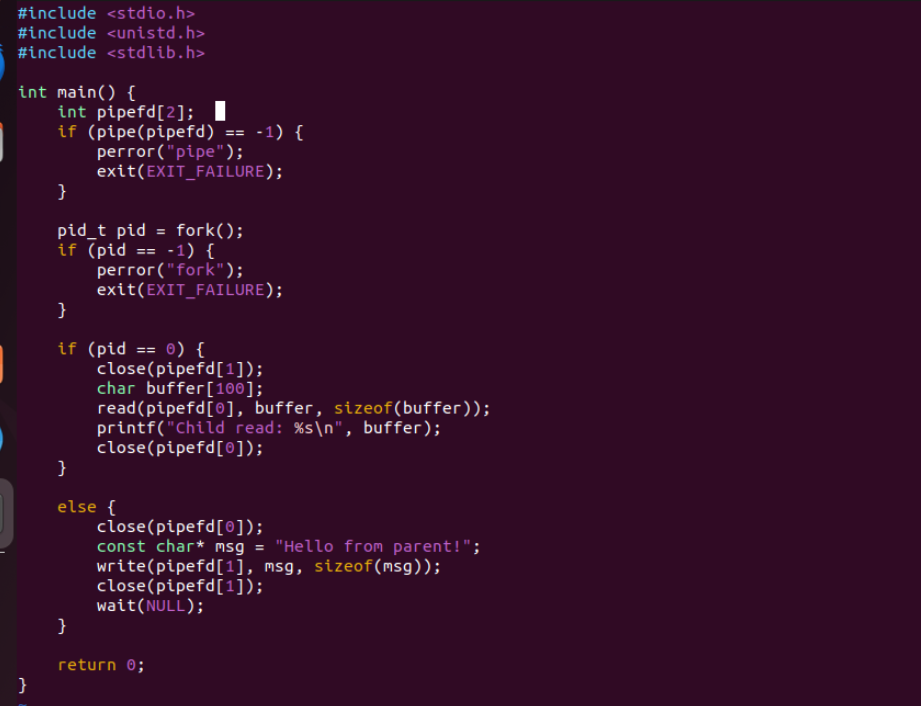
Shared Memory: Allocate and manage shared memory segments for efficient data sharing between multiple processes.

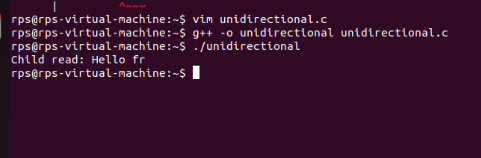
Message Queues: Create and utilize message queues for asynchronous communication and data exchange with message prioritization.

Synchronization: Implement appropriate synchronization mechanisms (e.g., semaphores, mutexes) to coordinate access to shared resources and prevent race conditions.

Error Handling: Incorporate robust error handling to manage potential IPC failures and resource leaks.

Unidirectional

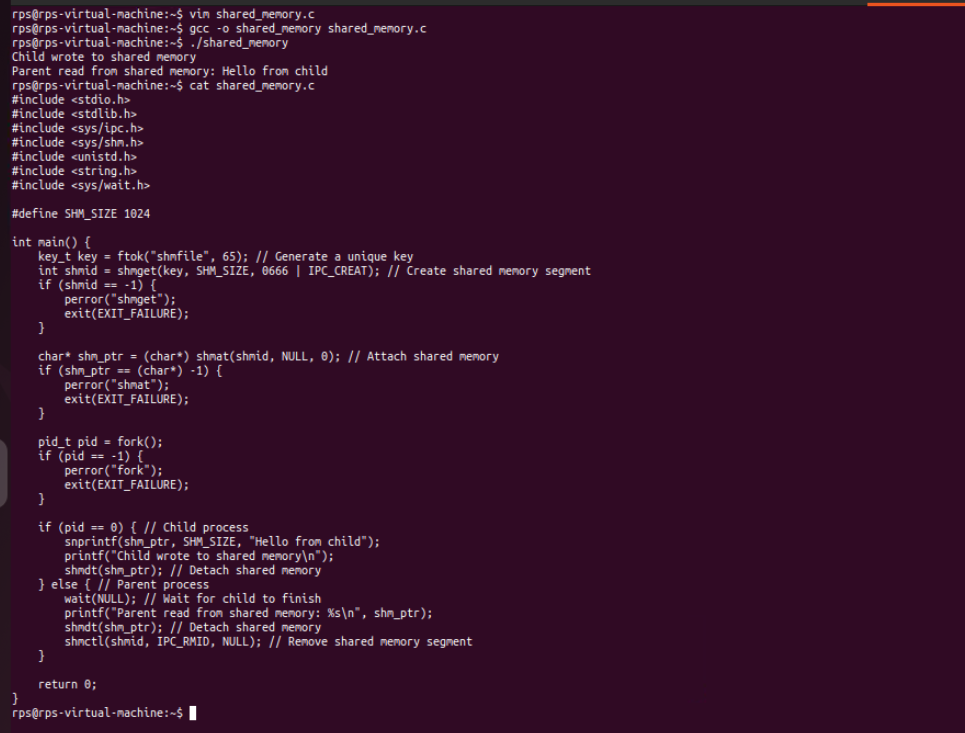




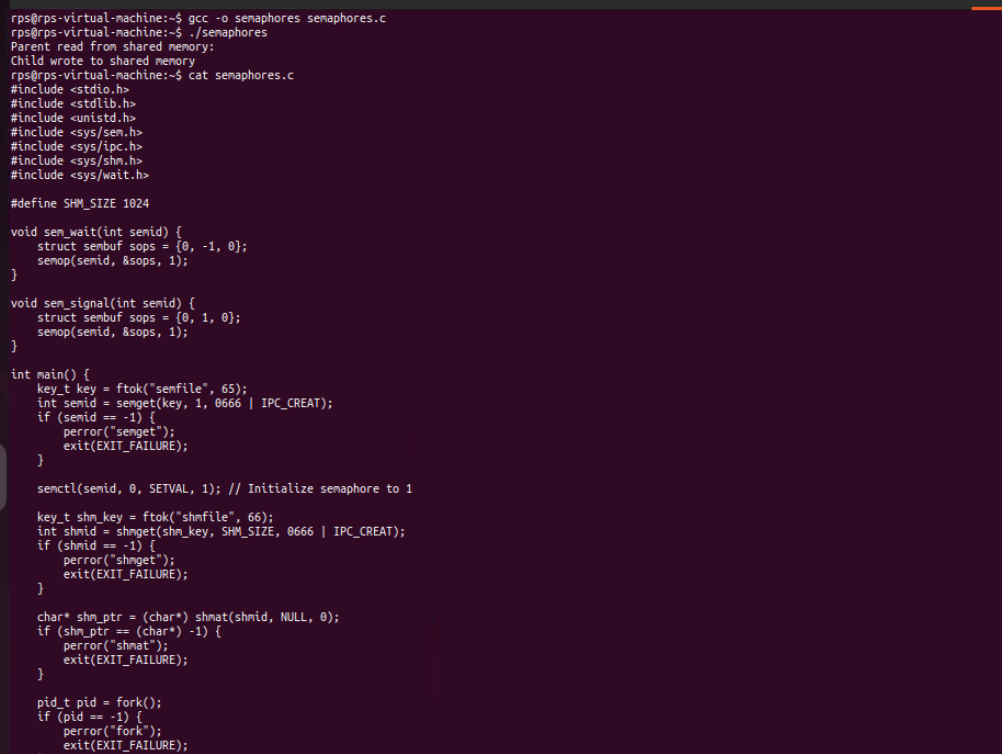
bidirectional

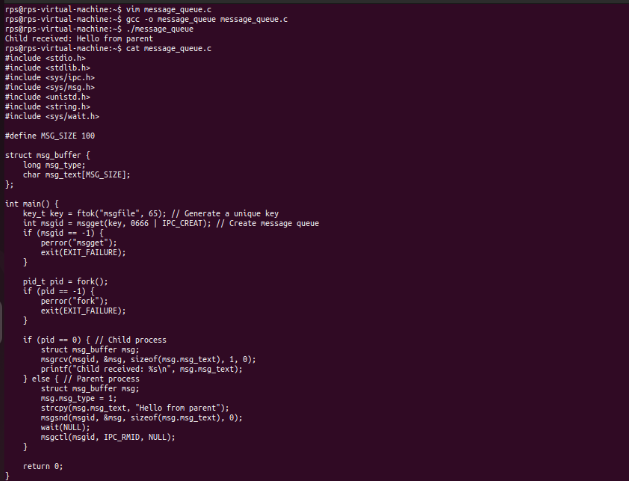


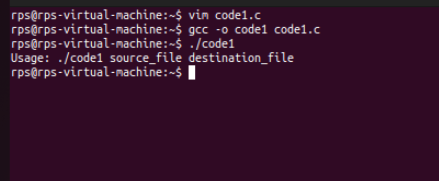
Shared\_memory



semaphores



Message\_queues

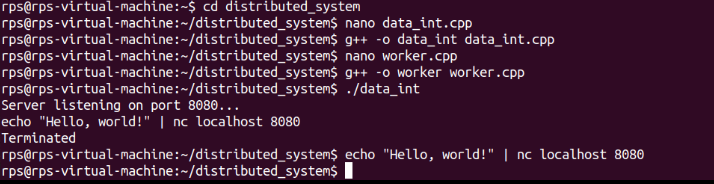


TEST:

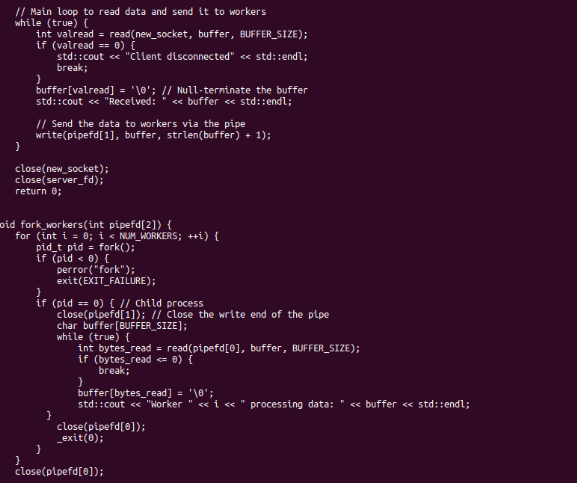
Design and implement a robust, distributed system using C++ that effectively leverages signals, sockets, and inter-process communication (IPC) to manage and coordinate multiple processes for a real-time data processing pipeline.

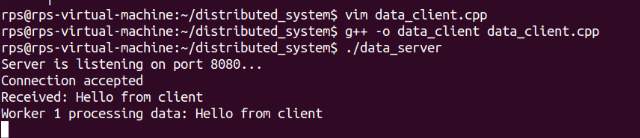
System Requirements

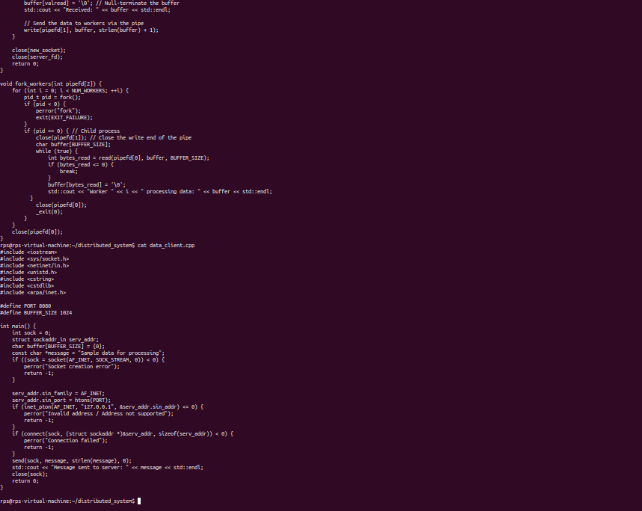
Data Ingestion: Continuously receive data from multiple sources (e.g., network sockets, files, sensors) and distribute it across multiple worker processes.

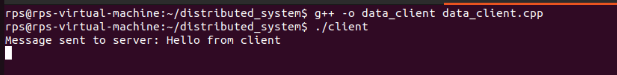


2. Data Processing: Distribute incoming data to multiple worker processes, each responsible for specific data transformations or calculations.

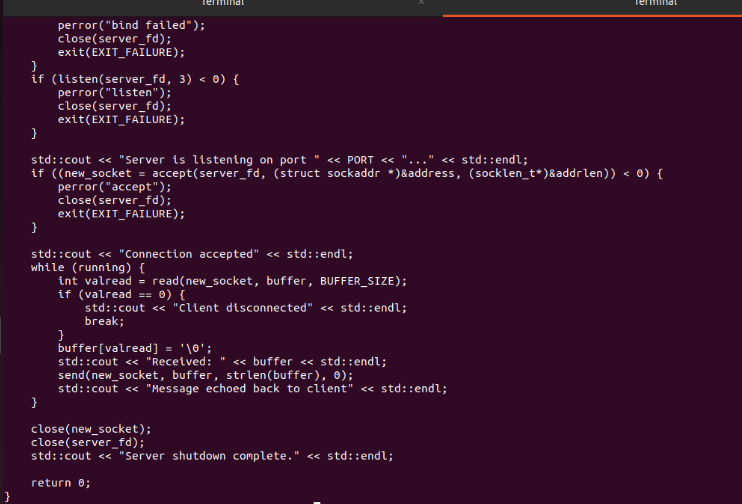


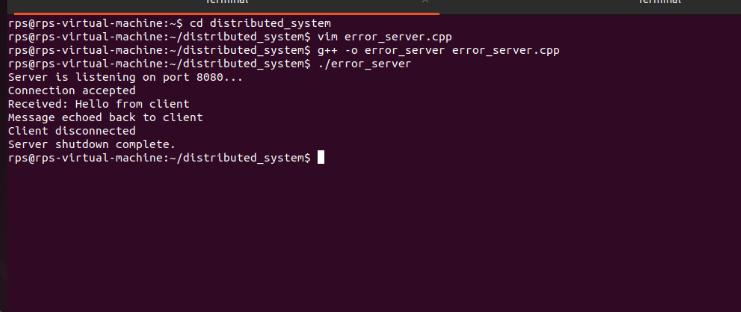


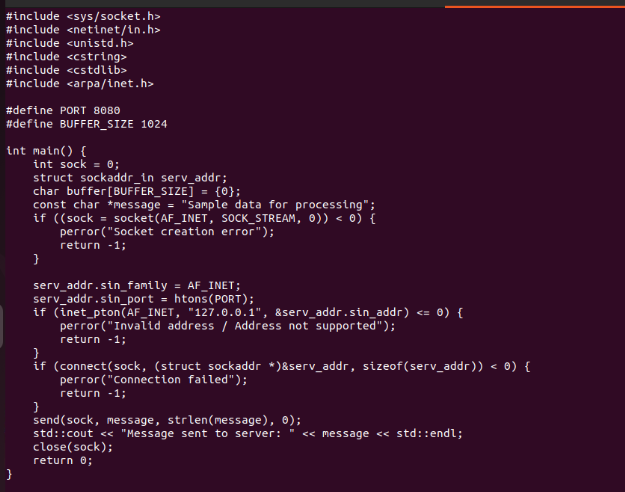


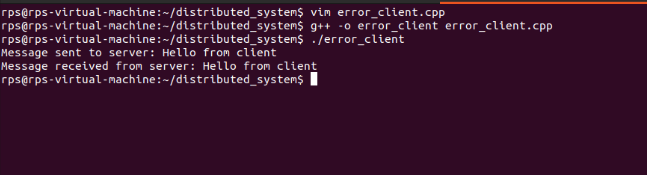


3. Error Handling: Implement robust error handling mechanisms using signals to gracefully handle unexpected events (e.g., process termination, network failures).









4. Inter-Process Communication: Utilize IPC (e.g., shared memory, message queues) for efficient communication and synchronization between processes.

5.Performance Optimization: Optimize the system for low latency and high throughput, considering factors like network congestion, process scheduling, and data transfer efficiency.

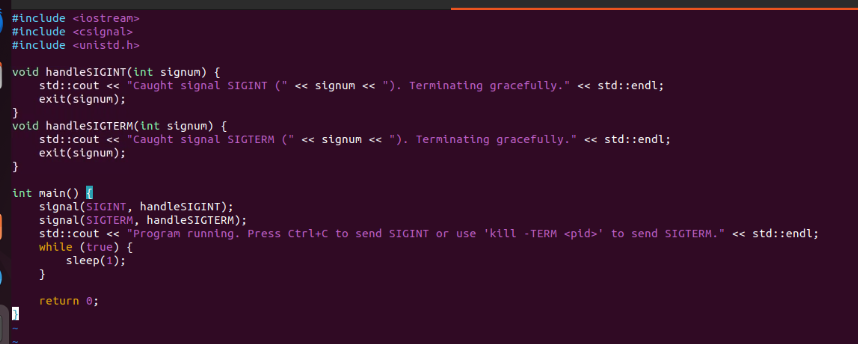
6. Scalability: Design the system to handle increasing data volumes and processing load by dynamically adjusting the number of worker processes.

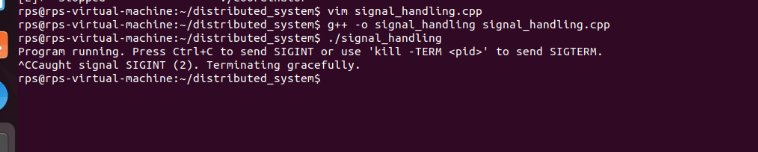
Coding Questions in C++

Signal Handling:

Write a C++ program that sets up a signal handler for SIGINT. The program should perform some tasks and print a message when SIGINT is caught, then terminate gracefully.

How would you modify your program to handle multiple different signals, each with a unique handling function?

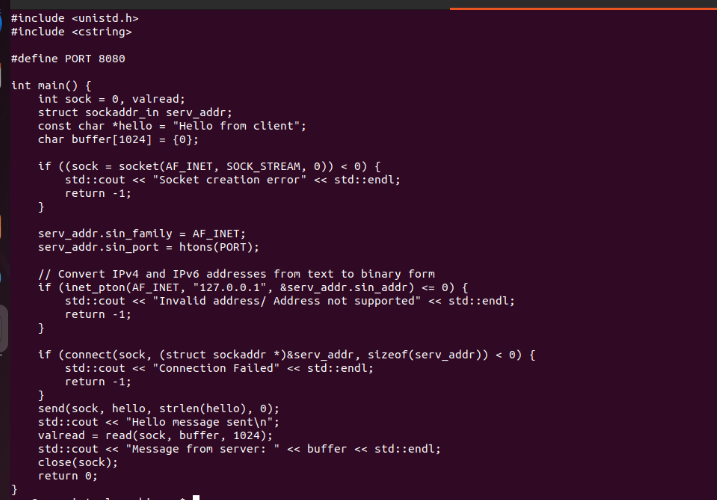




Sockets for Network Communication:

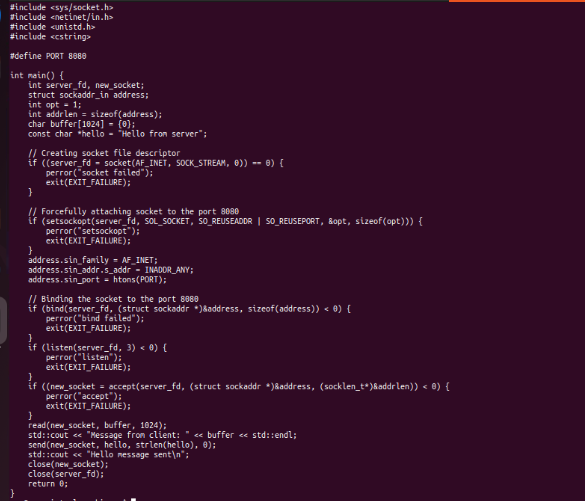
Implement a simple echo server in C++ that listens on a specific port, accepts client connections, and echoes back any messages received from clients.

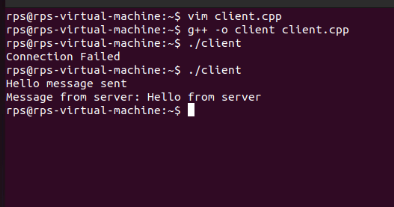
Write a client program that connects to the echo server, sends a message, and prints the echoed response.





Client



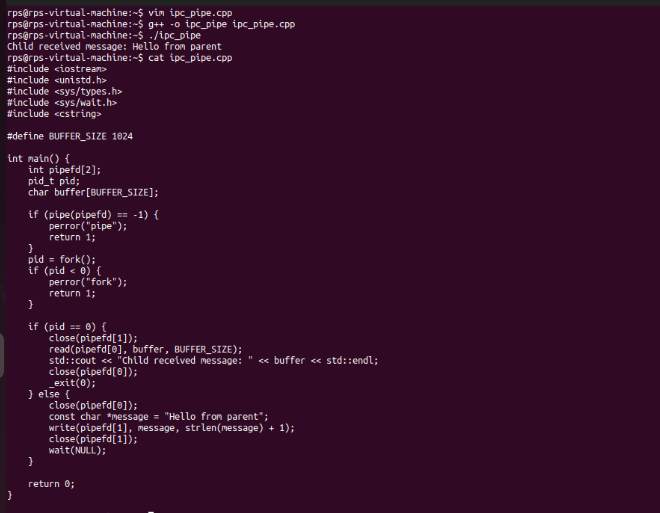


Inter-Process Communication (IPC):

Write a C++ program that creates a parent process and a child process. Use a pipe for IPC to send a message from the parent to the child, and have the child process print the message.

How would you modify the program to use a message queue instead of a pipe for communication between the parent and child processes?

Ipc pipe



Ipc\_queues

