DAY-27:

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.

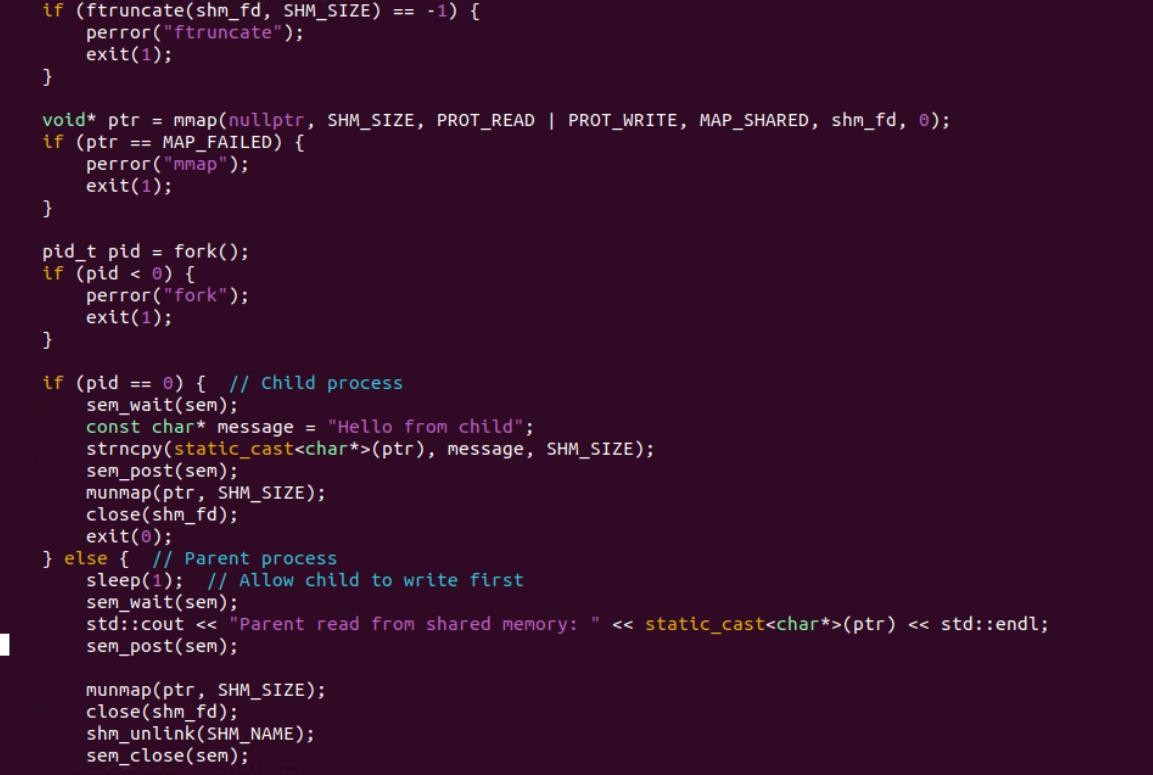
Error Handling: Incorporate robust error handling to manage potential IPC failures and resource leaks.

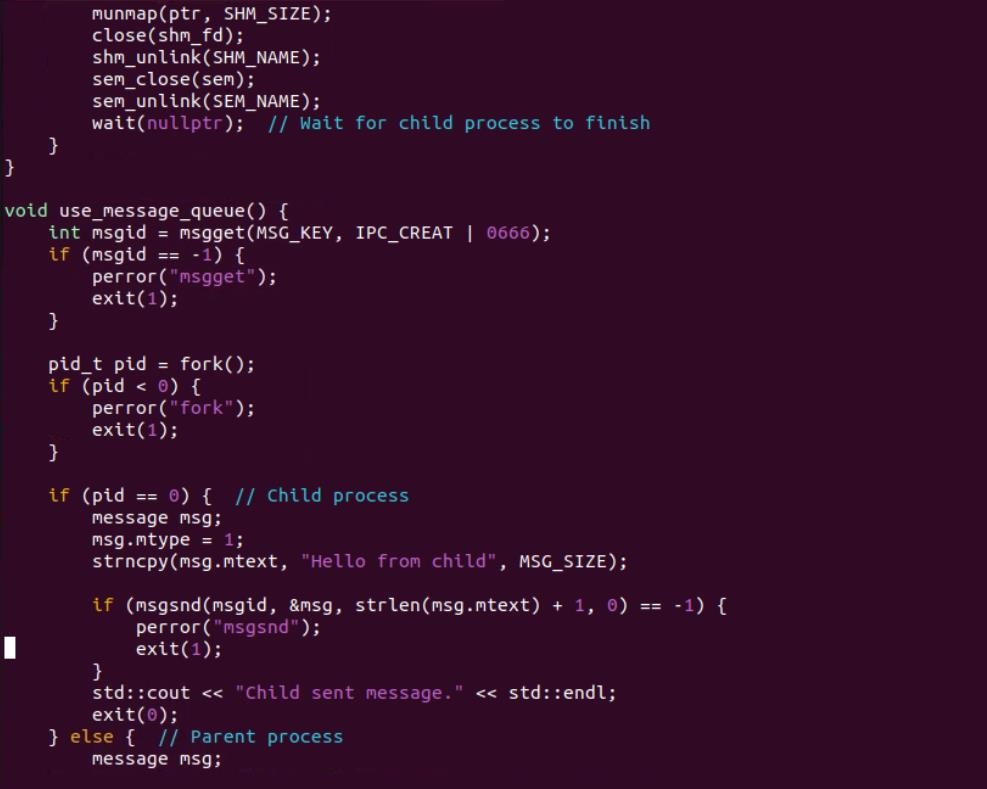
****

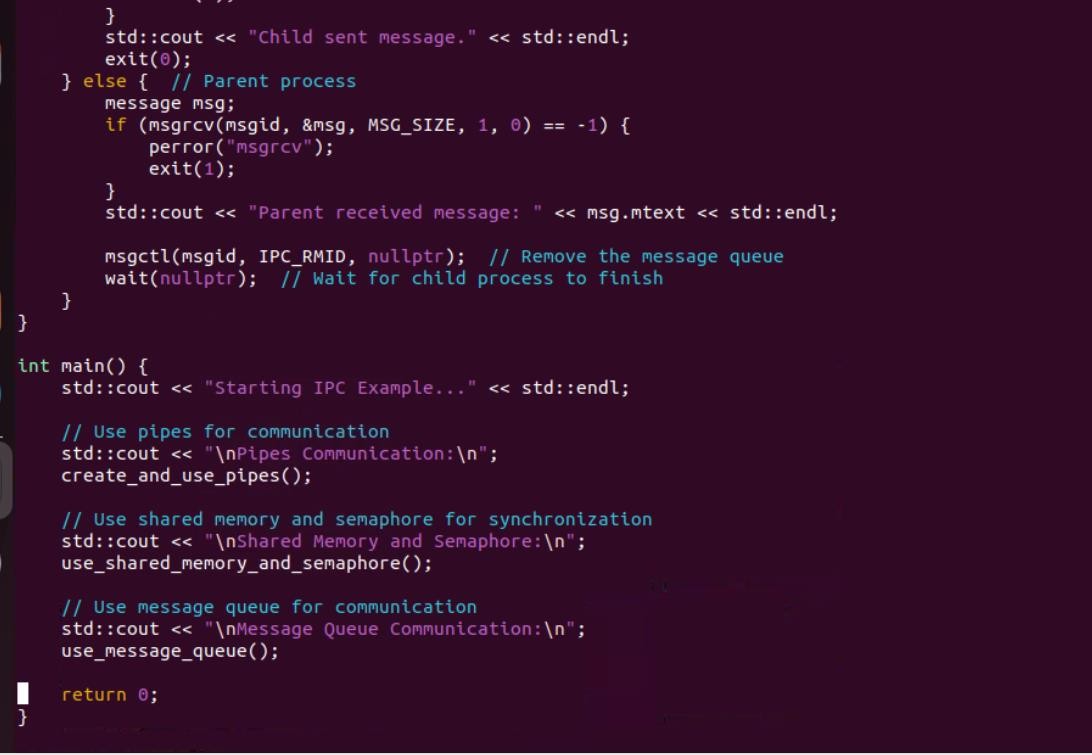
**Code:**



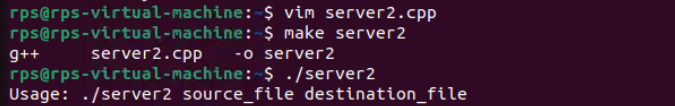
****

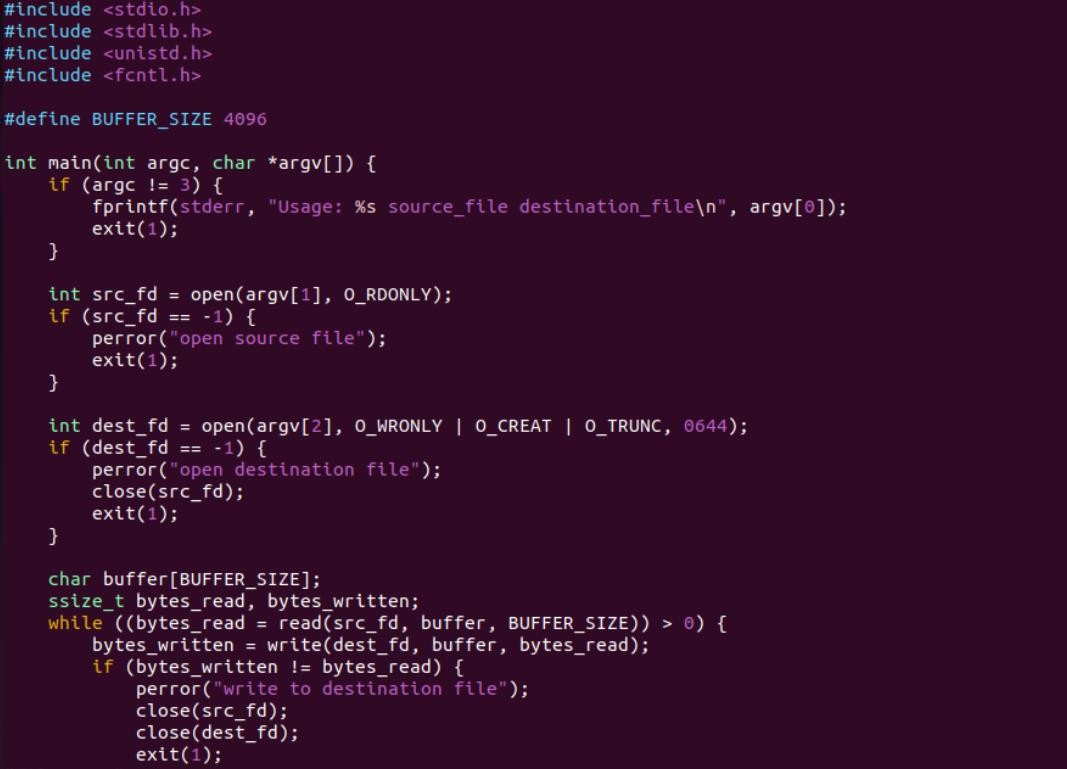


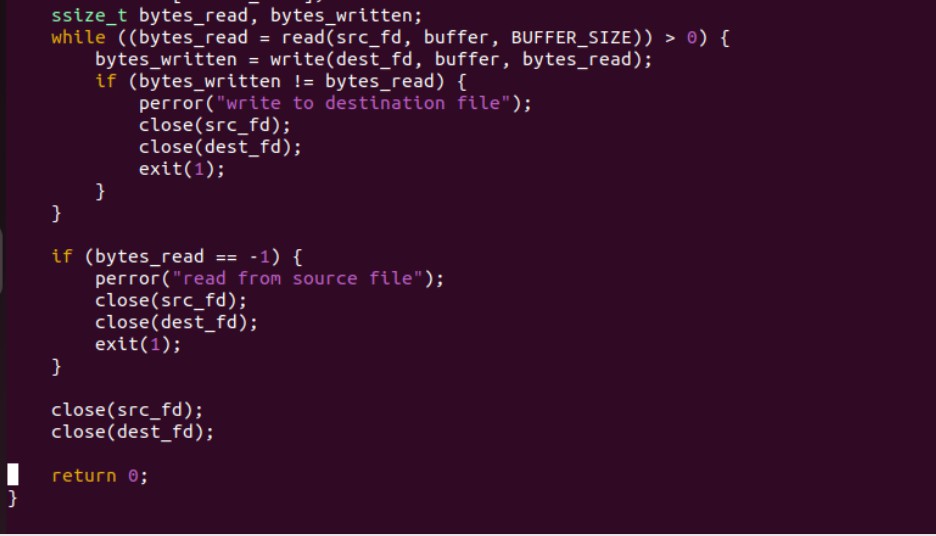
****



**CODE FOR SERVER:**

****

****

****

create a program that replicates the functionality of the standard cp command, but without using any

standard library functions related to file I/O. Instead, you must employ system calls directly to perform file operations.

Requirements:

System calls: Utilize system calls like open, close, read, and write to interact with files.

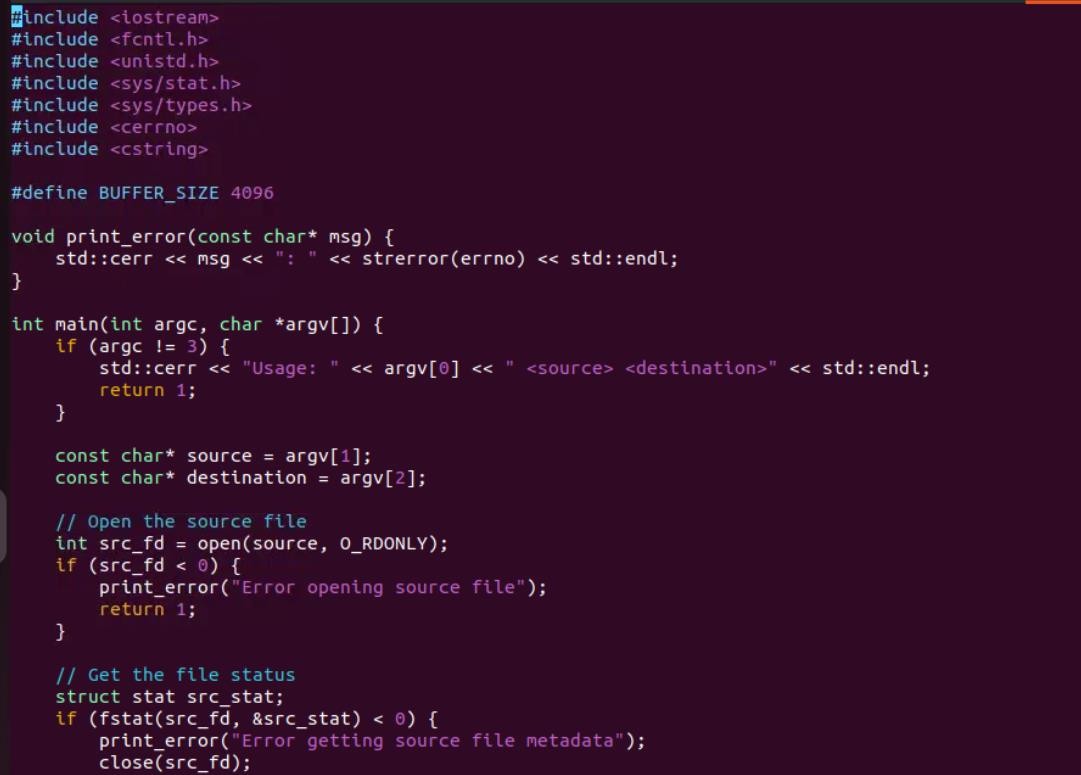
Error handling: Implement robust error handling for potential issues such as file not found, permission denied, disk full, etc.

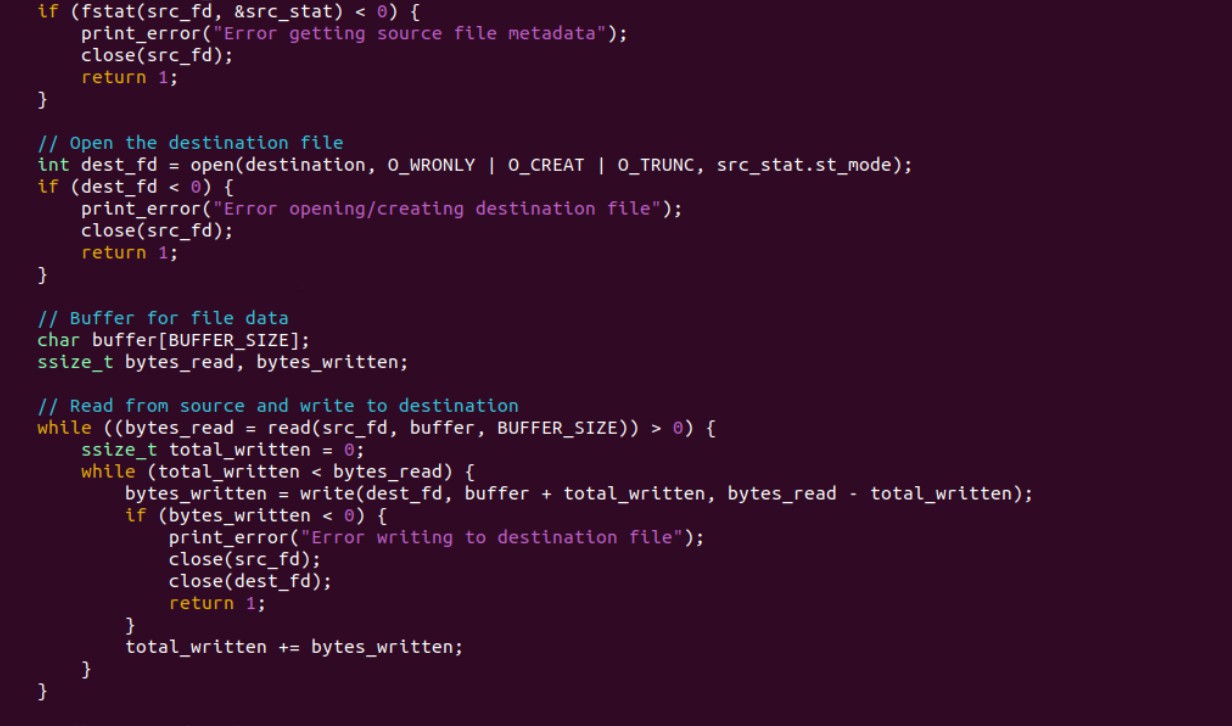
Efficiency: Optimize the copying process for performance, considering buffer sizes and read/write operations.

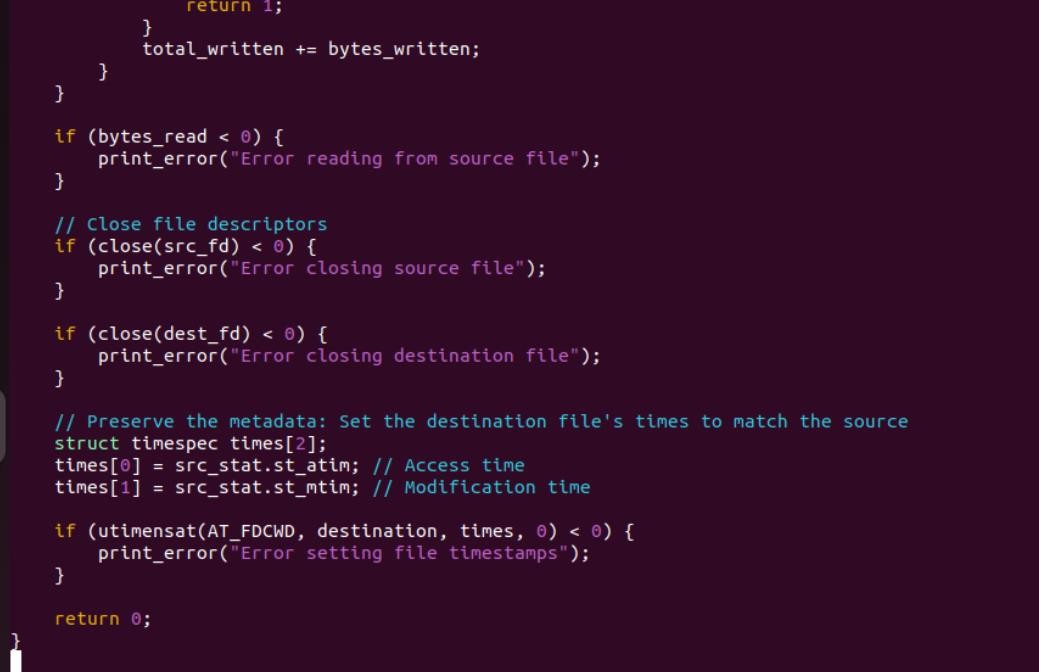
Metadata: Preserve file permissions, timestamps, and other relevant metadata during the copy process.

User interface: Provide a simple command-line interface with options for source and destination file paths.

CODE :-

****

****

****

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.

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

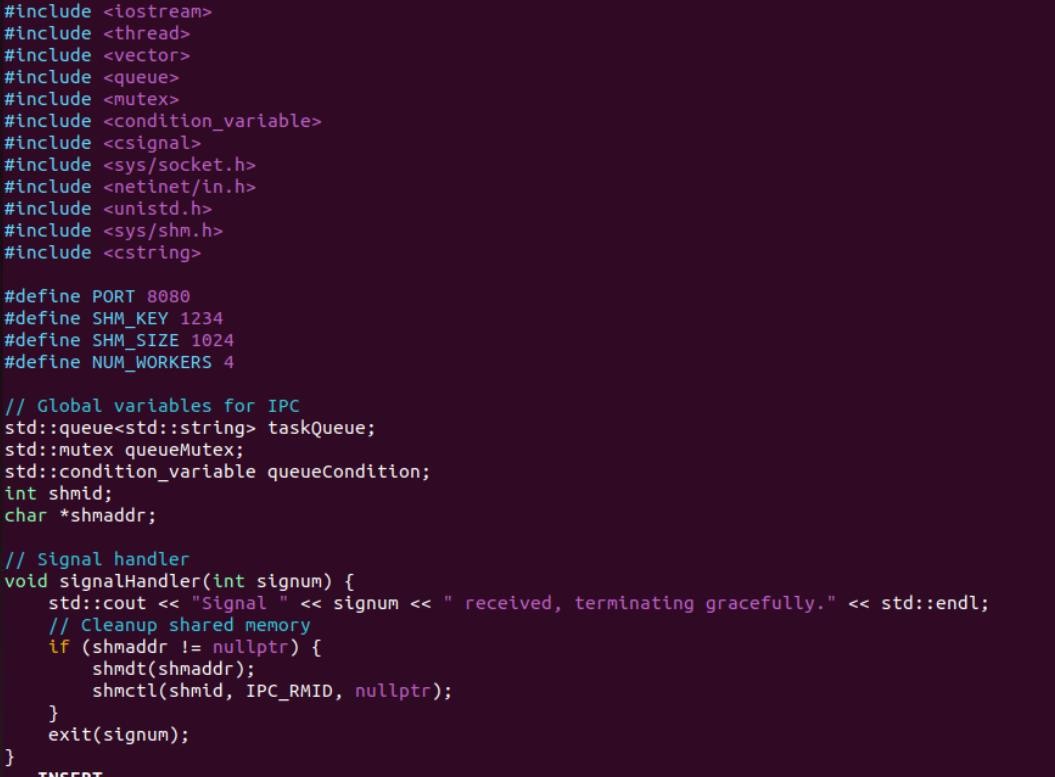
Error Handling: Implement robust error handling mechanisms using signals to gracefully handle unexpected events (e.g., process termination, network failures).

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

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

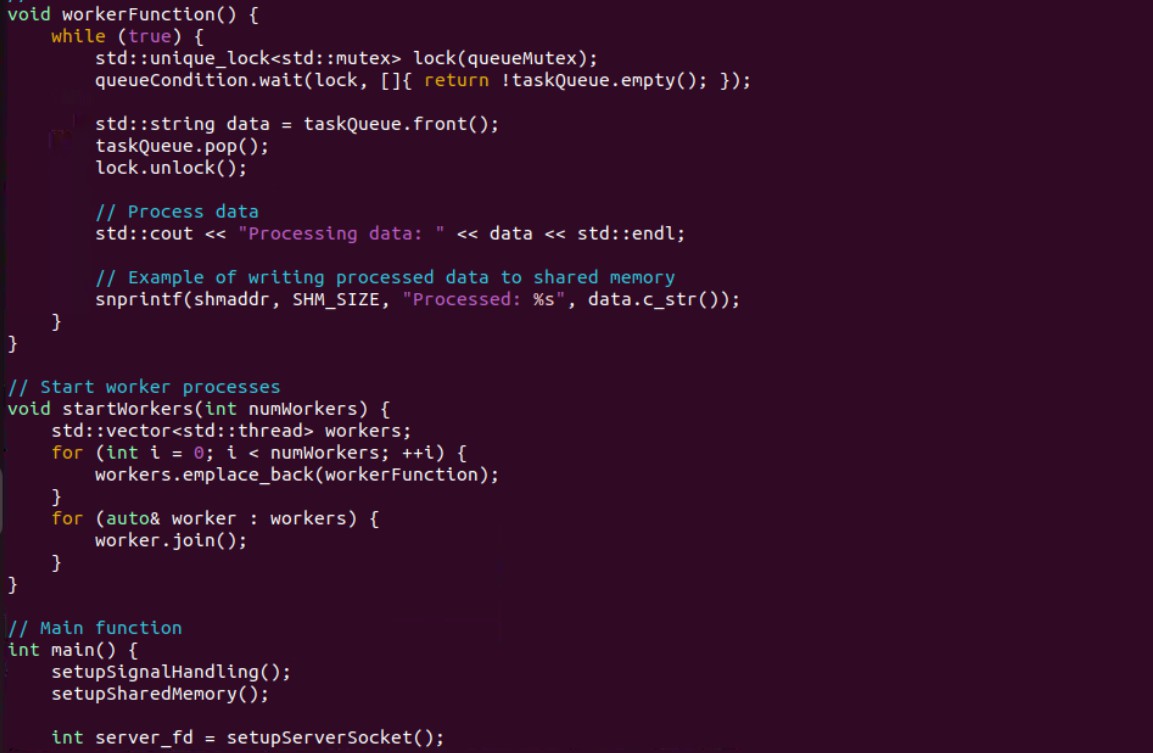
Scalability: Design the system to handle increasing data volumes and processing load by dynamically adjusting the number of worker processes.

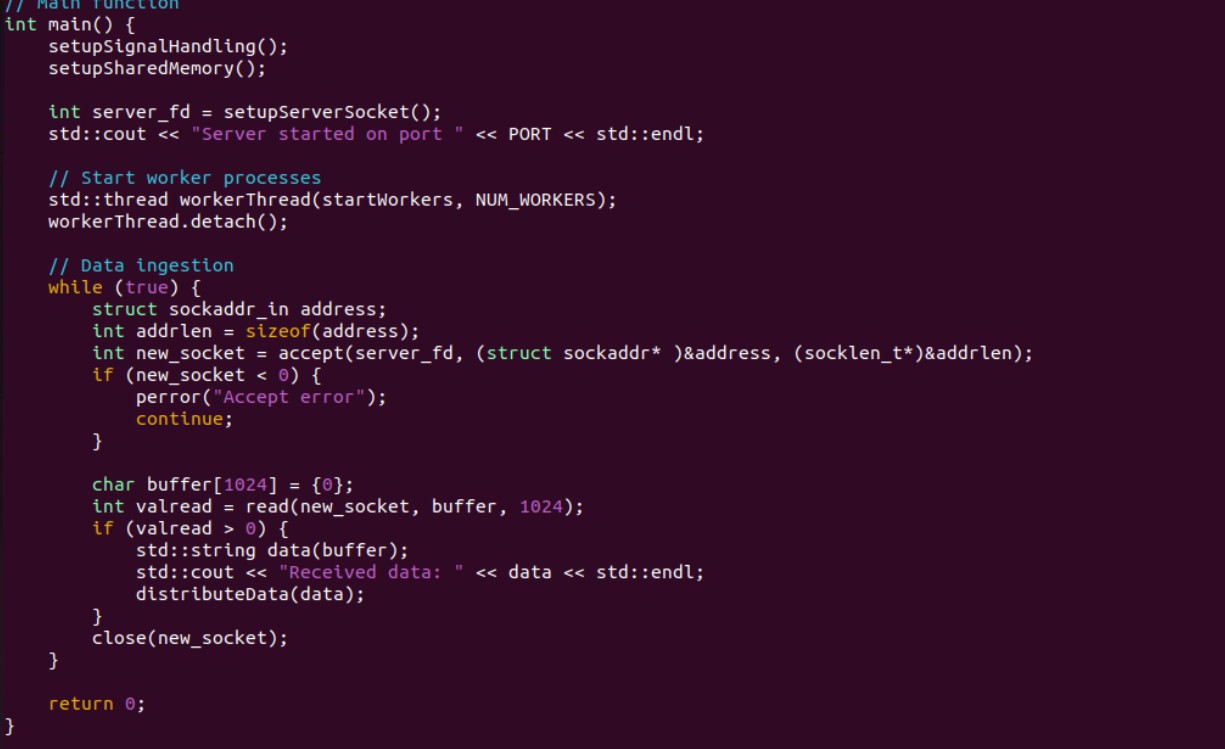
****

****

****

****

****

****