南京邮电大学

专业课程设计 I 报告

（ 2023 / 2024 学年 第 二 学期）

题　　目：客户/服务器程序的同步与通信机制的设计

专　　业	计算机科学与技术
学 生 姓 名	周才凯
班 级 学 号	B21030129
指 导 教 师	徐小龙
指 导 单 位	计算机科学与技术系
日　　期	**2024.6.17-2024.6.30**

支撑指标点	评价准则	计分（每项10分）
课程目标1：通过课程设计，培养学生综合应用操作系统等计算机技术等领域专业知识的技能。（20分）	1、能够掌握操作系统的相关基础知识，并能够针对求解的工程问题，收集资料进行合理的分析与设计	
	2、通过调研，能够选择合适的程序设计语言与编程开发平台，对求解的工程问题进行编程实现	
课程目标2：深入理解操作系统领域的相关知识，比较各种解决方案的优缺点，解决复杂工程问题的实践创新能力。（20分）	3、能够给出数据结构和算法的设计描述，给出关键算法的流程图或伪代码，并给出各算法之间的结构关系描述	
	4、具备一定的人机交互设计意识，人机交互设计合理、友好，操作简便	
课程目标3：能够解决进程通信中的具体问题，能够对操作系统中相关算法进行模拟，能够实现文件系统等部分模块的功能，培养工程实践能力。（30分）	5、对操作系统相关知识有一定的掌握，能够完成课题要求的各项任务和指标	
	6、能够结合计算机软硬件资源，合理选用算法、数据结构、数据存储方式等技术手段，理解相关算法，并进行模拟。	
	7、掌握调试方法与工具，对程序开发过程中出现的问题进行分析、跟踪与调试，并能够进行充分测试	
课程目标4：分组完成一次项目设计与开发的全过程，组内成员通过讨论和交流解决课程设计中的难题，了解课程设计的实现和具体操作系统中相应部分的实现两者之间区别。在实验报告中准确阐述课程设计的内容，能够针对具体工程问题指定解决方案，清晰陈述观点和回答问题。（30分）	8、组内成员之间有一定的团队合作，互通有无	
	9、能够正确、完整地回答指导教师关于课题的问询，反映其对课题内容，以及相关的工程基础知识具有较好的理解和掌握	
	10、具备一定的语言表达能力与文字处理能力，能够结合复杂工程问题撰写报告，报告内容和实验数据详实，格式规范	
专业课程设计I能力测评总分		
指导教师：_____ **_____年___月___日**		
备注：		

客户/服务器程序的同步与通信机制的设计

一、课题内容和要求

深入掌握 Linux 操作系统下的进程间同步、通信的相关方法；

设计一个具体的应用场景（如电子交易）和两个交互进程；

一个服务者进程和一个调用者进程，消息格式和内容自行设定；

通过显示结果分析程序的正确性；

对所采用的算法、程序结构和主要函数过程以及关键变量进行详细的说明；

提供关键程序的清单、源程序及可执行文件和相关的软件说明；

对程序的调试过程所遇到的问题进行回顾和分析，对测试和运行结果进行分析；

总结软件设计和实习的经验和体会，进一步改进的设想。

二、课题需求分析

1、本课题目标系统“股票查询交易系统”的功能框架图如图 1 所示。

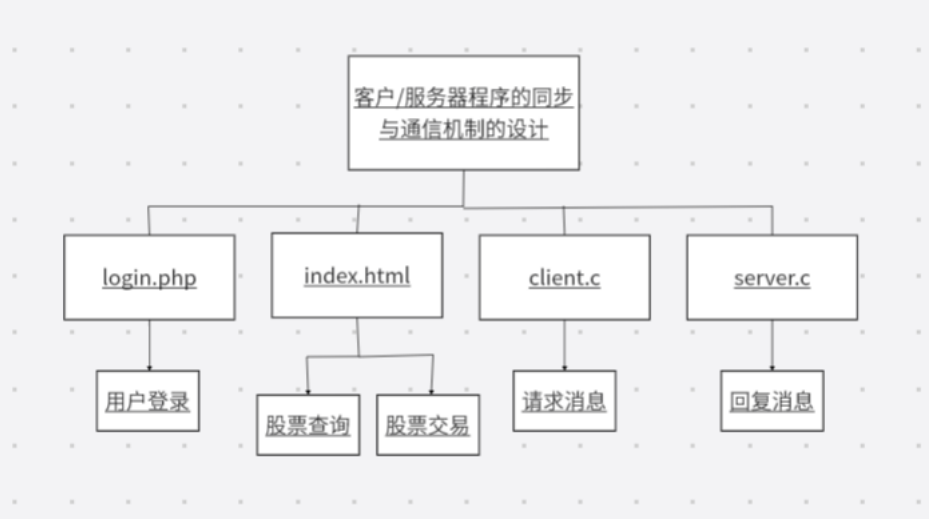

图 1 功能框架图

（1）通过 microhttpd 实现小型服务器；

（2）利用 fetch（）实现 index.html 和 client.c 的交互；

（3）利用 socket（）实现 server.c 和 client.c 的交互；

（4）支持对不同股票代码的查询交易。

（5）设计了用户登录验证功能。

2、需求分析

①应用场景

电子交易系统：用户可以查询股票信息和进行股票交易。

服务器端：负责存储用户账户信息、股票信息和处理交易请求。

客户端：提供用户界面，允许用户登录、查询股票信息和提交交易请求。

客户端包括 login.php index.html client.c

②功能需求

用户登录：用户输入用户名和密码，系统验证后登录。

股票查询：用户可以查询特定股票的当前价格和历史价格。（历史价格还没做）

股票交易：用户可以提交买入或卖出股票的请求。

服务器端处理：服务器端处理股票查询和交易请求，更新股票信息和用户账户状态。

③性能需求

实时性：股票信息和交易结果需要实时更新。

可靠性：系统需要保证数据的一致性和完整性。

安全性：用户数据和交易信息需要加密保护。（没做）

④用户界面需求

用户友好：界面简洁直观，易于操作。

响应式：界面需要适应不同设备的屏幕尺寸。（没做）

⑤系统设计

进程间同步与通信机制

进程间通信：使用 socket 进行进程间通信。

同步机制：使用互斥锁（mutex）和条件变量（condition variable）来同步进程间的操作。（将交易进程上锁，防止不同用户交易导致数据不一致）

⑥系统架构

客户端：前端使用 HTML/CSS/JavaScript 实现用户界面，后端使用 socket 与服务器通信。

服务器端：使用 C++语言实现，负责处理客户端请求和管理数据。

3、实现过程

①服务器端实现

进程管理：创建服务者进程监听客户端连接请求。

消息处理：定义消息格式，处理查询和交易请求。

数据同步：使用同步机制确保数据一致性。（锁）

②客户端实现

用户界面：使用 HTML/CSS/JavaScript 实现用户登录、股票查询和交易界面。

通信逻辑：使用 JavaScript 的 fetch API 与服务器通信。

③HTML 页面实现

登录页面：实现用户登录表单，提交后跳转到 index.html。

主页面：展示股票信息和交易表单。

④关键程序清单

服务器端：server.cpp

客户端：index.html、login.html、client.cpp。

⑤测试与调试

单元测试：对服务器端的各个功能模块进行单元测试。

集成测试：测试客户端与服务器端的通信和同步机制。

性能测试：确保系统在高负载下仍能稳定运行。（maybe）

⑥总结与改进

总结：回顾项目开发过程中的关键决策和遇到的问题。

改进：根据测试结果和用户反馈，提出改进方案。

三、课题相关数据结构及算法设计

1 主要数据结构

①股票库存管理

std::map<std::string, int> inventory = {{"商品 A", 10}, {"商品 B", 5}};

inventory：使用 std::map 来存储商品名称和库存数量。键（key）是商品名称（std::string 类型），值（value）是库存数量（int 类型）。这种结构允许快速查找商品库存。

②用户认证信息

std::map<std::string, std::string> user_credentials = {{"用户 1", "密码 1"}, {"用户 2", "密码

2"}};

user_credentials：使用 std::map 来存储用户名称和密码。键是用户名称（std::string 类型），值是密码（std::string 类型）。这种结构允许快速验证用户身份，便于遍历和批量操作。

2 主要算法流程

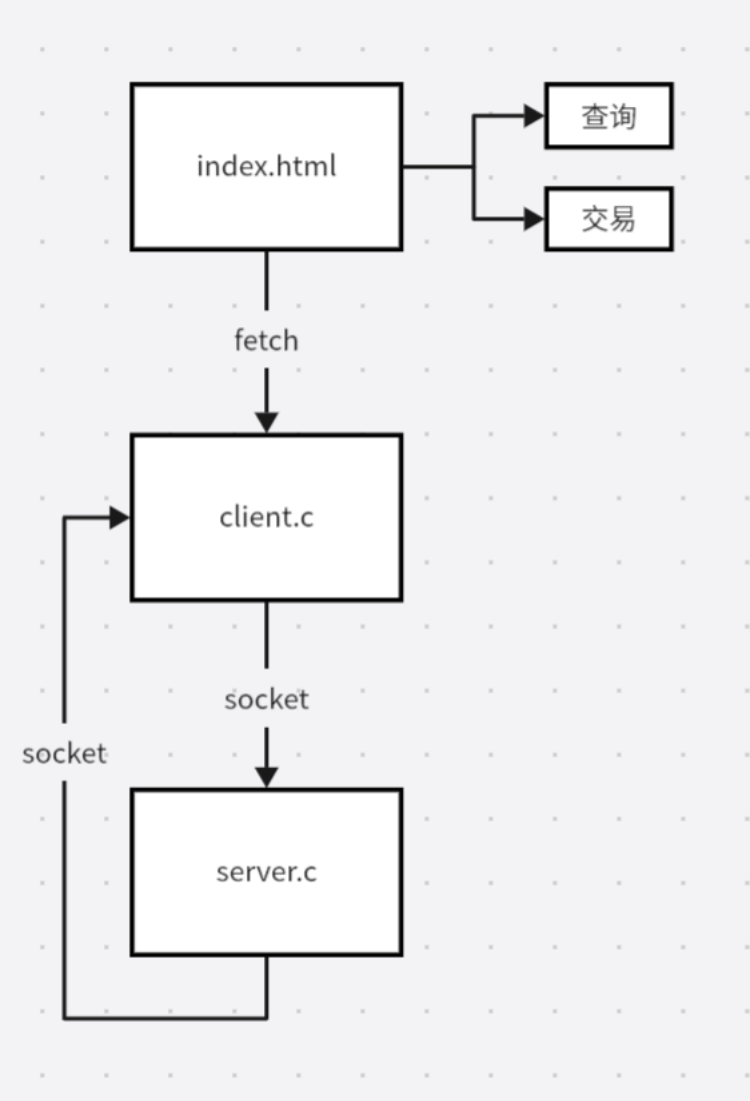

图 2 流程图简化

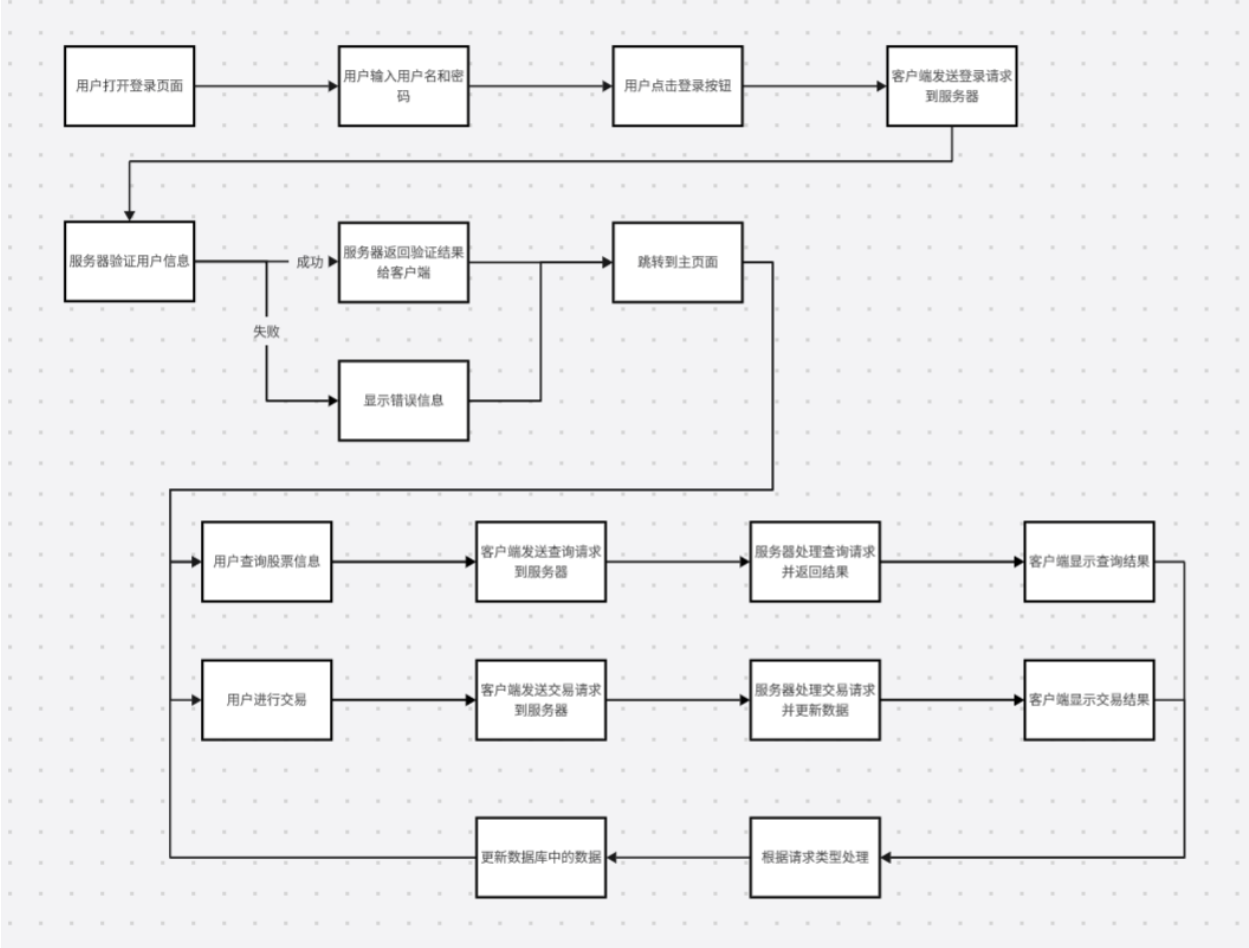

图 3 流程图详细

1.商品库存管理算法：

查询库存：通过商品名称在 std::map 中查找对应的库存数量。
更新库存：根据商品名称和数量更新库存。如果商品不存在，则添加新商品及其库存数量。
库存检查：在更新库存前，检查库存数量是否足够，以避免超卖。
库存报告：生成库存报告，列出所有商品的名称和库存数量。
库存补充：根据库存报告，补充库存不足的商品。

2.用户认证算法：
查询库存：通过商品名称在 std::map 中查找对应的库存数量。
更新库存：根据商品名称和数量更新库存。如果商品不存在，则添加新商品及其库存数量。
库存检查：在更新库存前，检查库存数量是否足够，以避免超卖。
库存报告：生成库存报告，列出所有商品的名称和库存数量。
库存补充：根据库存报告，补充库存不足的商品。

3.服务器端算法
初始化 socket：创建一个 socket，指定使用 IPv4 地址和 TCP 协议。
绑定地址：将 socket 绑定到一个 IP 地址和端口号上。
监听连接：设置 socket 为监听模式，准备接受客户端的连接请求。
接受连接：等待客户端的连接请求，接受连接后建立连接。

数据接收：从连接的 socket 中读取数据。
数据处理：对接收到的数据进行处理。
数据发送：将处理后的数据发送回客户端。
关闭连接：通信完成后关闭 socket 连接。
释放资源：关闭 socket，释放相关资源。

4. 客户端算法
初始化 socket：创建一个 socket，指定使用 IPv4 地址和 TCP 协议。
连接服务器：指定服务器的 IP 地址和端口号，尝试与服务器建立连接。
数据发送：将数据发送到服务器。
数据接收：从服务器接收响应数据。
关闭连接：通信完成后关闭 socket 连接。
释放资源：关闭 socket，释放相关资源。

四、源程序代码

4-1 不包含 html 的版本

```
//ser1.cpp
#include <iostream>
#include <string>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

// 定义服务器监听的端口号
const int PORT = 12345;

// 处理客户端请求的函数
std::string handle_request(const std::string& request) {
    // 如果请求以"QUERY:"开头
    if (request.find("QUERY:") == 0) {
        // 提取股票代码
        std::string stock_symbol = request.substr(6); // 6 表示从 request 字符串的第7个字符开始提取子串（因为字符串索引从 0 开始）
        // 返回模拟的股票价格查询响应
        if(stock_symbol=="GOOGL"){
            return "RESPONSE:Price of " + stock_symbol + " is $200";
        }
        return "RESPONSE:Price of " + stock_symbol + " is $100";
    }
    // 如果请求以"TRADE:"开头
```

```
        else if (request.find("TRADE:") == 0) {
            // 查找第一个冒号的位置
            size_t first_colon = request.find(":", 6); // find 函数的第二个参数 6 是一个起始搜索位置。这意味着 find 函数将从 request 字符串的第 7 个字符开始搜索（因为字符串索引从 0 开始），直到字符串的末尾
            // 提取股票代码
            std::string stock_symbol = request.substr(6, first_colon - 6);
            // 提取交易数量
            std::string quantity = request.substr(first_colon + 1);
            // 返回模拟的交易响应
            return "RESPONSE:Traded " + quantity + " of " + stock_symbol;
        }
        // 如果请求不符合预期格式
        else {
            // 返回无效请求的响应
            return "RESPONSE:Invalid request";
        }
    }

    // 服务器主函数
    void server_main() {
        // 创建 socket 文件描述符
        int server_fd, new_socket;
        // 定义服务器地址结构体
        struct sockaddr_in address;

    // struct sockaddr_in {
    //     sa_family_t    sin_family; // 地址族，对于 IPv4 应该设置为 AF_INET
    //     uint16_t       sin_port;   // 端口号，使用 htons() 函数转换为网络字节序
    //     struct in_addr sin_addr;   // IPv4 地址，使用 inet_addr() 或 inet_aton() 函数填充
    //     char           sin_zero[8]; // 保留空间，通常填充为 0
    // };

        // 设置 socket 选项
        int opt = 1;
        // 设置地址长度变量
        int addrlen = sizeof(address);
        // 定义用于接收数据的缓冲区
        char buffer[1024] = {0};

        // 1、创建 socket
        if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
            perror("socket failed");
```

```
        exit(EXIT_FAILURE);
    }

    // 2、设置 socket 选项，允许重用地址和端口
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt)))
{
        perror("setsockopt");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 设置服务器地址结构体的属性
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY; // 允许任何 IP 地址连接
    address.sin_port = htons(PORT); // 将端口号转换为网络字节序

    // 3、绑定 socket 到指定的地址和端口
    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 4、开始监听连接
    if (listen(server_fd, 3) < 0) { // 3 表示允许的最大连接数
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // 打印服务器监听端口信息
    std::cout << "Server listening on port " << PORT << "..." << std::endl;

    // 5、无限循环，等待客户端连接
    while (true) {
        // 接受客户端连接
        if ((new_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen)) < 0) {
            perror("accept");
            close(server_fd);
            exit(EXIT_FAILURE);
        }

        // 6、数据传输
```

```
            // 从客户端读取数据
            int valread = read(new_socket, buffer, 1024);
            // 将接收到的数据转换为字符串
            std::string request(buffer, valread);
            // 处理请求并获取响应
            std::string response = handle_request(request);
            // 发送响应给客户端
            send(new_socket, response.c_str(), response.length(), 0);
            // 7、关闭与客户端的连接
            close(new_socket);
            // close(server_fd); // 关闭监听 socket
        }
}

// 主函数，启动服务器
int main() {
        server_main();
        return 0;
}

//cli1.cpp
#include <iostream>
#include <string>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

// 定义服务器端口
const int PORT = 12345;

// 客户端主函数
void client_main() {
        // 创建一个 socket 文件描述符
        int sock = 0;
        // 定义服务器地址结构体
        struct sockaddr_in serv_addr;
        // 定义用于接收数据的缓冲区
        char buffer[1024] = {0};

        // 1、创建 socket，返回文件描述符
```

```
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::cerr << "Socket creation error" << std::endl;
        return;
    }

    // 2、设置服务器地址结构体的属性
    serv_addr.sin_family = AF_INET; //  使用 IPv4 地址
    serv_addr.sin_port = htons(PORT); //  将端口号转换为网络字节序

    //  将服务器地址转换为网络字节序
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address/ Address not supported" << std::endl;
        return;
    }

    // 3、连接到服务器
    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cerr << "Connection Failed" << std::endl;
        return;
    }

    // 4、数据传输
    //  从标准输入读取用户请求
    std::string request;
    std::cout << "Enter your request (e.g., QUERY:GOOGL or TRADE:GOOGL:10): ";
    std::getline(std::cin, request);

    //  发送请求到服务器
    send(sock, request.c_str(), request.length(), 0);
    //  从服务器接收响应
    int valread = read(sock, buffer, 1024);
    //  输出服务器的响应
    std::cout << "Server response: " << std::string(buffer, valread) << std::endl;

    // 5、关闭 socket
    close(sock);
}

//  主函数，调用客户端主函数
int main() {
    client_main();
    return 0;
}
```

4-2 包含 html 的版本

```
<!DOCTYPE html> <!-- 声明文档类型为 HTML5 -->
<html> <!-- 页面的根元素 -->
<head> <!-- 页面的头部信息 -->
    <title>Stock Trading System</title> <!-- 页面标题 -->
    <script> <!-- 页面内嵌的 JavaScript 代码 -->
        // 定义一个名为 queryStock 的函数，用于查询股票信息
        function queryStock() {
            // 获取页面上 id 为 stockCode 的输入框的值
            var stockCode = document.getElementById("stockCode").value;
            // 使用 fetch API 发送一个 GET 请求到服务器的 /query 路径，并将股票代码作为查询参数
            fetch(`/query?code=${stockCode}`)
                // 处理响应，将响应转换为文本格式
                .then(response => response.text())
                // 将响应的文本内容设置为页面上 id 为 result 的元素的文本内容
                .then(data => {
                    document.getElementById("result").innerText = data;
                });
        }

        // 定义一个名为 tradeStock 的函数，用于执行股票交易
        function tradeStock() {
            // 获取页面上 id 为 stockCode 的输入框的值
            var stockCode = document.getElementById("stockCode").value;
            // 获取页面上 id 为 quantity 的输入框的值
            var quantity = document.getElementById("quantity").value;
            // 使用 fetch API 发送一个 POST 请求到服务器的 /trade 路径，并将股票代码和数量作为查询参数
            fetch(`/trade?code=${stockCode}&quantity=${quantity}`)
                // 处理响应，将响应转换为文本格式
                .then(response => response.text())
                // 将响应的文本内容设置为页面上 id 为 result 的元素的文本内容
                .then(data => {
                    document.getElementById("result").innerText = data;
                });
        }
    </script> <!-- 结束内嵌的 JavaScript 代码 -->
</head> <!-- 结束页面的头部信息 -->

<body> <!-- 页面的主体内容 -->
    <h1>Stock Trading System</h1> <!-- 页面的标题 -->
    <label for="stockCode">Stock Code:</label> <!-- 标签，提示用户输入股票代码 -->
```

```
    <input type="text" id="stockCode" name="stockCode"> <!-- 输入框，用于输入股票代码 -->
    <br><br> <!-- 换行 -->
    <label for="quantity">Quantity:</label> <!-- 标签，提示用户输入交易数量 -->
    <input type="number" id="quantity" name="quantity"> <!-- 输入框，用于输入交易数量 -->
    <br><br> <!-- 换行 -->
    <!-- 按钮，点击时调用 queryStock 函数，查询股票信息 -->
    <button onclick="queryStock()">Query Stock</button>
    <!-- 按钮，点击时调用 tradeStock 函数，执行股票交易 -->
    <button onclick="tradeStock()">Trade Stock</button>
    <h2>Result:</h2> <!-- 结果标题 -->
    <p id="result"></p> <!-- 结果显示区域 -->
</body> <!-- 结束页面的主体内容 -->
</html> <!-- 结束 HTML 页面 -->

//client.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <unistd.h>
#include <microhttpd.h>

#define PORT 8888
#define QUEUE_NAME "/stock_queue"
#define RESPONSE_QUEUE_NAME "/stock_response_queue"
#define MAX_SIZE 1024

// 定义股票消息结构体
struct StockMessage {
    long type; // 消息类型
    char stock_code[10]; // 股票代码
    int quantity; // 交易数量
};

// 定义股票响应结构体
struct StockResponse {
    long type; // 响应类型
    char response[MAX_SIZE]; // 响应内容
};

// 发送消息到队列的函数
```

```
void sendMessage(long type, const char* stockCode, int quantity, char* result) {
    mqd_t mq, mq_response; //  消息队列和响应队列的句柄
    struct StockMessage msg; //  股票消息结构体
    struct StockResponse res; //  股票响应结构体

    //  打开股票消息队列
    mq = mq_open(QUEUE_NAME, O_WRONLY);
    if (mq == -1) {
        perror("Client: mq_open (stock_queue)");
        strcpy(result, "Error opening stock queue");
        return;
    }

    //  准备发送的消息
    msg.type = type;
    strncpy(msg.stock_code, stockCode, sizeof(msg.stock_code) - 1);
    msg.stock_code[sizeof(msg.stock_code) - 1] = '\0';
    msg.quantity = quantity;

    //  发送消息到队列
    if (mq_send(mq, (char*)&msg, sizeof(msg), 0) == -1) {
        perror("Client: mq_send");
        strcpy(result, "Error sending message");
        return;
    }

    //  关闭消息队列
    mq_close(mq);

    //  打开响应队列
    mq_response = mq_open(RESPONSE_QUEUE_NAME, O_RDONLY);
    if (mq_response == -1) {
        perror("Client: mq_open (stock_response_queue)");
        strcpy(result, "Error opening response queue");
        return;
    }

    //  从响应队列接收响应
    if (mq_receive(mq_response, (char*)&res, sizeof(res), NULL) == -1) {
        perror("Client: mq_receive");
        strcpy(result, "Error receiving response");
        return;
    }
```

```
        // 关闭响应队列
        mq_close(mq_response);

        // 将响应复制到结果字符串
        strcpy(result, res.response);
    }

    // 处理 HTTP 连接的函数
    int answer_to_connection(void* cls, struct MHD_Connection* connection,
        const char* url, const char* method, const char* version,
        const char* upload_data, size_t* upload_data_size, void** con_cls) {
        const char* page = "<html><body>Request not supported</body></html>"; // 默认页面
        struct MHD_Response* response; // HTTP 响应
        int ret; // 返回值

        char result[MAX_SIZE] = { 0 }; // 用于存储响应结果的缓冲区

        // 检查请求方法是否为 GET
        if (strcmp(method, "GET") != 0)
            return MHD_NO;

        // 根据 URL 处理不同的请求
        if (strncmp(url, "/query", 6) == 0) {
            const char* stockCode = MHD_lookup_connection_value(connection,
MHD_GET_ARGUMENT_KIND, "code");
            if (stockCode) {
                sendMessage(1, stockCode, 0, result); // 发送查询请求
            }
        } else if (strncmp(url, "/trade", 6) == 0) {
            const char* stockCode = MHD_lookup_connection_value(connection,
MHD_GET_ARGUMENT_KIND, "code");
            const char* quantityStr = MHD_lookup_connection_value(connection,
MHD_GET_ARGUMENT_KIND, "quantity");
            if (stockCode && quantityStr) {
                int quantity = atoi(quantityStr);
                sendMessage(2, stockCode, quantity, result); // 发送交易请求
            }
        }

        // 创建 HTTP 响应
        response = MHD_create_response_from_buffer(strlen(result), (void*)result,
MHD_RESPMEM_PERSISTENT);
```

```
    // 发送 HTTP 响应
    ret = MHD_queue_response(connection, MHD_HTTP_OK, response);
    // 销毁响应对象
    MHD_destroy_response(response);

    return ret;
}

int main() {
    struct MHD_Daemon* daemon; // HTTP 服务器守护进程

    // 启动 HTTP 服务器守护进程
    daemon = MHD_start_daemon(MHD_USE_SELECT_INTERNALLY, PORT, NULL, NULL,
        &answer_to_connection, NULL, MHD_OPTION_END);
    if (NULL == daemon) return 1;

    // 打印服务器运行信息
    printf("Server running on port %d\n", PORT);

    // 等待用户输入，防止程序立即退出
    getchar();

    // 停止 HTTP 服务器守护进程
    MHD_stop_daemon(daemon);
    return 0;
}

//server.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <unistd.h>

// 定义消息队列名称
#define QUEUE_NAME "/stock_queue"
#define RESPONSE_QUEUE_NAME "/stock_response_queue"
#define MAX_SIZE 1024
#define STOCK_QUERY 1
#define STOCK_TRADE 2
```

```
// 定义股票消息结构体
struct StockMessage {
    long type; // 消息类型，用于区分查询和交易
    char stock_code[10]; // 股票代码
    int quantity; // 交易数量
};

// 定义股票响应结构体
struct StockResponse {
    long type; // 响应类型，与消息类型对应
    char response[MAX_SIZE]; // 响应内容
};

// 处理股票查询请求的函数
void handle_query(struct StockMessage* msg, struct StockResponse* res) {
    // 假设股票价格为 100 美元，实际应用中应从数据库或 API 获取
    sprintf(res->response, "Stock %s price is $100", msg->stock_code);
}

// 处理股票交易请求的函数
void handle_trade(struct StockMessage* msg, struct StockResponse* res) {
    // 假设交易成功，实际应用中应进行交易逻辑处理
    sprintf(res->response, "Traded %d shares of %s", msg->quantity, msg->stock_code);
}

int main() {
    mqd_t mq, mq_response; // 消息队列和响应队列的句柄
    struct mq_attr attr; // 消息队列属性
    struct StockMessage msg; // 用于接收消息的结构体
    struct StockResponse res; // 用于发送响应的结构体

    // 初始化消息队列属性
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10; // 最大消息数
    attr.mq_msgsize = sizeof(struct StockMessage); // 每个消息的最大字节数
    attr.mq_curmsgs = 0; // 当前消息数

    // 打开或创建消息队列
    mq = mq_open(QUEUE_NAME, O_CREAT | O_RDONLY, 0644, &attr);
    if (mq == -1) {
        perror("Server: mq_open (stock_queue)"); // 打开消息队列失败
        exit(1);
    }
```

```
    // 打开或创建响应队列
    mq_response = mq_open(RESPONSE_QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
    if (mq_response == -1) {
        perror("Server: mq_open (stock_response_queue)"); // 打开响应队列失败
        exit(1);
    }

    // 服务器主循环，处理消息队列中的消息
    while (1) {
        // 从消息队列中接收消息
        if (mq_receive(mq, (char*)&msg, sizeof(msg), NULL) == -1) {
            perror("Server: mq_receive"); // 接收消息失败
            continue;
        }

        // 根据消息类型处理请求
        res.type = msg.type;
        if (msg.type == STOCK_QUERY) {
            handle_query(&msg, &res); // 处理查询请求
        } else if (msg.type == STOCK_TRADE) {
            handle_trade(&msg, &res); // 处理交易请求
        }

        // 将响应发送到响应队列
        if (mq_send(mq_response, (char*)&res, sizeof(res), 0) == -1) {
            perror("Server: mq_send"); // 发送响应失败
        }
    }

    // 关闭消息队列和响应队列
    mq_close(mq);
    mq_close(mq_response);

    // 删除消息队列和响应队列
    mq_unlink(QUEUE_NAME);
    mq_unlink(RESPONSE_QUEUE_NAME);

    return 0; // 程序正常退出
}
```

五、测试数据及其结果分析

对算法功能、性能以及健壮性评测所使用的输入数据进行介绍和说明，并给出与输入

数据相匹配的算法执行结果，并进行分析。

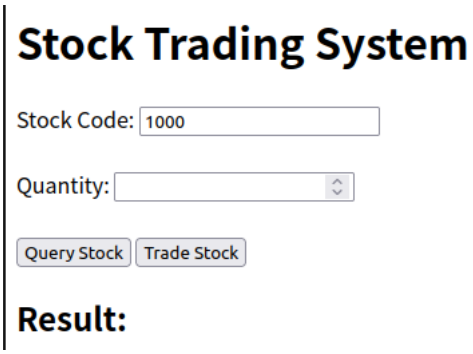

图 3 html 页面

1、进行股票查询

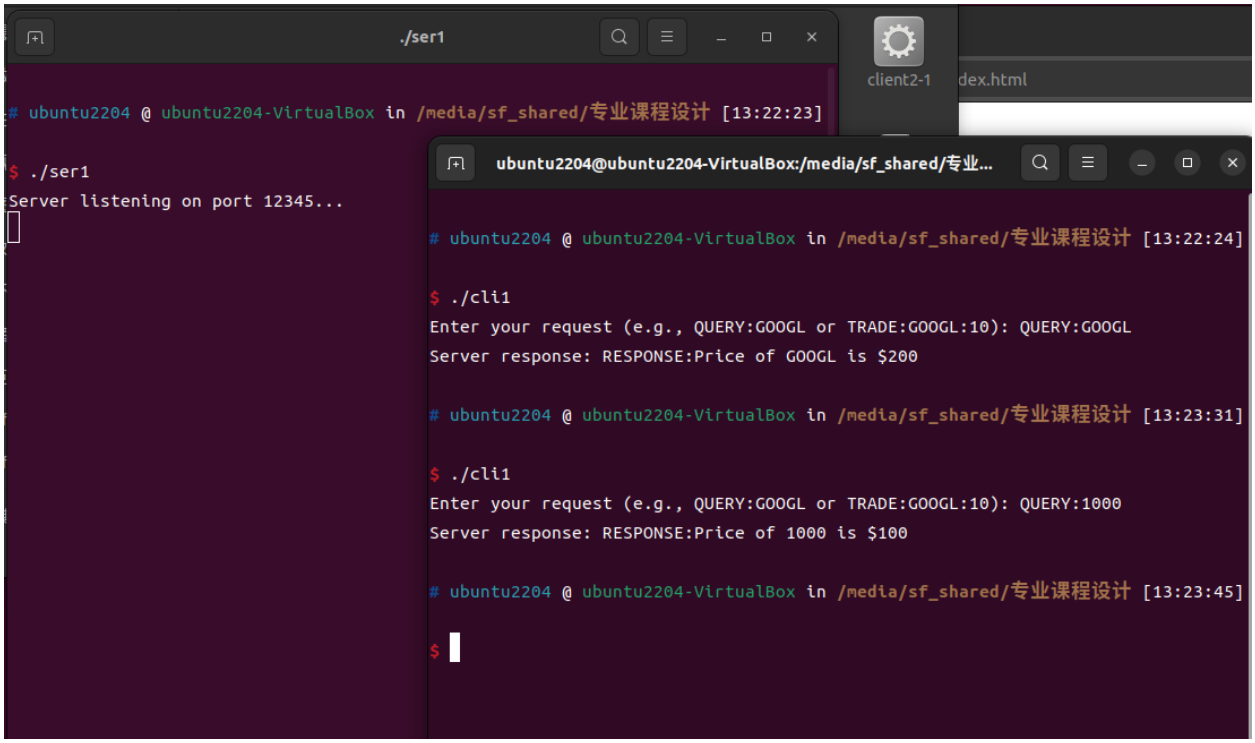

图 4 股票查询

分别输入 QUERY:GOOGL 和 QUERY : 1000；

返回 Price of GOOGL is $200 和 Price of 1000 is $100；

2、进行股票交易

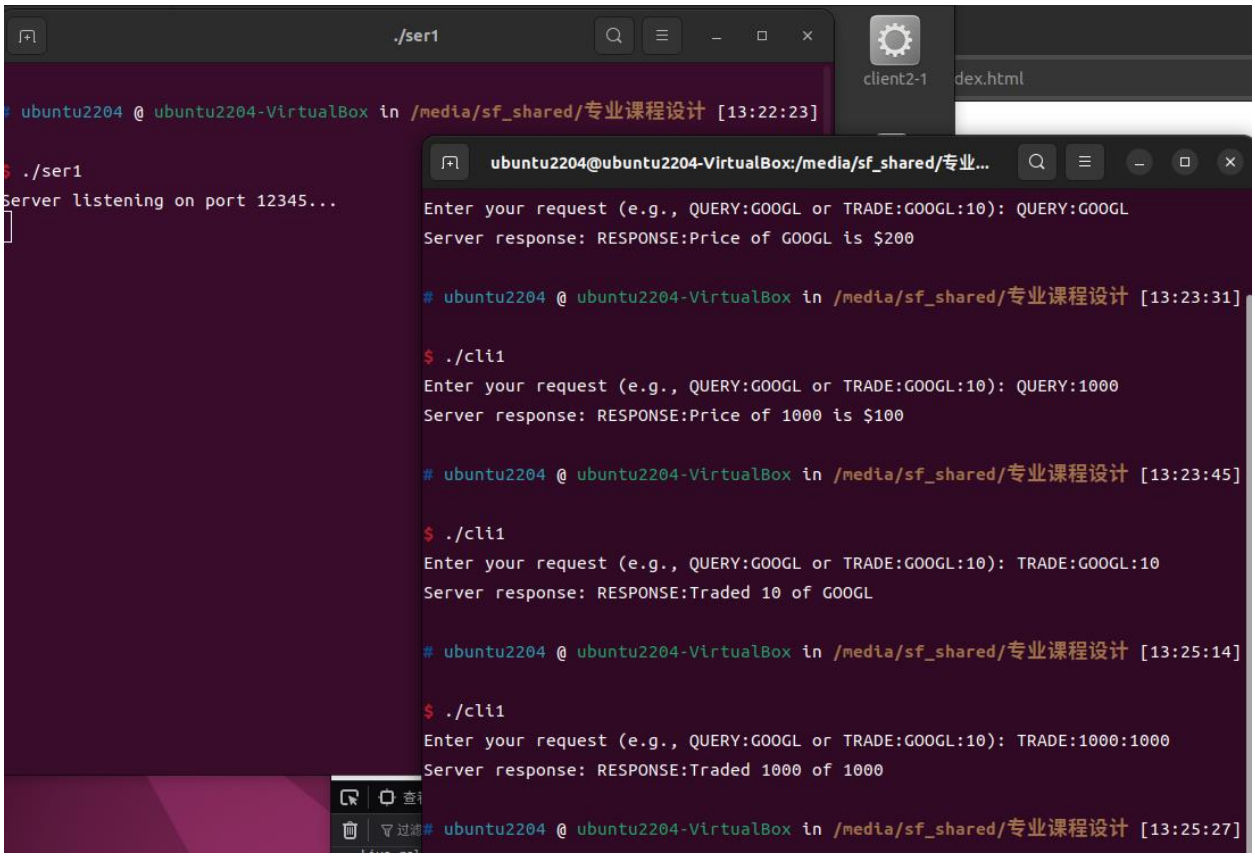

图 5 股票交易

输入 TRADE:GOOGL:10，返回 server response: RESPONSE:Traded 10 of GOOGL；

输入 TRADE:1000:1000，返回 server response: RESPONSE:Traded 1000 of 1000；

六、课题完成过程中遇到的问题及解决方法

在完成课题的过程中，遇到的问题及解决方法是项目成功的关键。

问题 1：在开发过程中，发现服务端和客户端之间的通信延迟较高，影响了用户体验。

解决方法：首先，通过网络分析工具（如 Wireshark）对通信过程进行监控，找出延迟的瓶颈。如果发现是由于网络延迟导致，可以考虑优化网络配置或使用更高效的通信协议。如果延迟是由于服务器处理请求的时间过长，可以对服务端代码进行优化，比如使用异步处理、优化数据库查询等方法来提高响应速度。

问题 2：在多线程环境下，发现共享资源的访问存在竞态条件，导致数据不一致。

解决方法：引入互斥锁（mutex）或读写锁（rwlock）来同步对共享资源的访问。在访问共享资源前，线程必须先获取锁，访问完毕后释放锁。这样可以确保在任何时刻，只有一个线程能够修改共享资源，从而避免数据不一致的问题。

问题 3：在测试阶段，发现程序在高并发情况下出现崩溃。

解决方法：首先，使用压力测试工具（如 Apache JMeter）模拟高并发场景，找出导致程序崩溃的原因。可能的原因包括内存泄漏、资源竞争、死锁等。针对这些问题，可以采取相应的措施，比如优化内存管理、改进同步机制、避免死锁等。此外，还可以通过增加服务器资源（如 CPU、内存）或优化系统架构来提高系统的并发处理能力。

问题 4：在用户界面设计中，发现界面布局在不同设备上显示效果不一致。

解决方法：采用响应式设计原则，确保界面布局能够适应不同屏幕尺寸和分辨率。可以使用 CSS 媒体查询（Media Queries）来根据不同的屏幕尺寸调整布局和样式。同时，使用相对单位（如 em、rem）而不是绝对单位（如 px），以确保元素的大小能够根据屏幕大小自适应。

问题 5：在项目开发后期，发现文档资料不完整，难以进行代码维护和后续开发。

解决方法：建立完善的文档编写和维护流程。在开发过程中，及时记录设计决策、代码实现细节和使用说明。使用版本控制系统（如 Git）来管理文档的变更历史，确保文档的完整性和可追溯性。此外，还可以通过编写 API 文档、代码注释和开发指南来提高代码的可读性和可维护性。

通过上述问题的解决方法，可以有效提升课题的开发效率和产品质量。 每个问题的解决都需要结合具体情况进行分析和调整，不断优化和改进，以确保课题的顺利完成。

七、总结

在设计一个客户/服务器程序时，深入理解 Linux 操作系统下的进程间同步与通信机制是至关重要的。Linux 提供了丰富的 IPC 机制，每种机制都有其独特的应用场景和优缺点。例如，管道适用于父子进程间的简单通信，而套接字则适合于网络环境中的进程间通信。在设计电子交易系统时，考虑到系统的复杂性和网络环境，套接字成为首选的通信方式。

在具体的应用场景设计中，电子交易系统需要处理股票查询和交易请求，同时确保交易的安全性和一致性。金钱信息存储在服务器端，服务者进程负责处理这些请求，而调用者进程则负责与用户交互。为了保证数据的一致性和操作的原子性，可以采用互斥锁和条件变量来同步进程间的操作。互斥锁用于保护共享资源，防止并发访问导致的数据不一致；条件变量则用于在资源未就绪时阻塞进程，直到资源可用。

程序的正确性验证是软件开发中的关键环节。通过编写详尽的测试用例，可以模拟各种情况下的请求，确保服务者进程能够正确处理查询和交易请求，并返回预期的结果。测试过程中，可以使用单元测试、集成测试和压力测试等多种测试方法，以全面评估程序的

性能和稳定性。

在程序调试过程中，可能会遇到死锁、资源竞争等并发问题。这些问题的解决需要深入理解进程同步机制和操作系统原理。通过使用调试工具和日志记录，可以追踪问题的根源并进行修复。调试过程中的经验教训对于提升软件质量和开发效率至关重要。

通过这个项目，我深刻体会到了软件设计的复杂性和挑战性。在设计和实现过程中，我学会了如何将理论知识应用到实际问题中，如何进行系统分析和设计，以及如何进行代码编写和调试。我认识到了团队合作的重要性，以及在项目开发过程中沟通和协作的重要性。这些经验对于我未来的职业生涯具有重要的指导意义。

为了进一步改进，我计划在未来的项目中采用更先进的设计模式和架构，比如微服务架构，以提高系统的可扩展性和可维护性。同时，我也会更加注重代码的可读性和可测试性，以提高代码的质量和开发效率。在实现用户界面方面，我将利用现代前端技术栈，如 React 或 Vue.js，结合后端服务，构建一个响应式、用户友好的界面。

总结来说，客户/服务器程序的设计是一个涉及多方面知识和技能的复杂过程。通过深入理解 Linux 下的 IPC 机制，设计合理的同步与通信机制，编写详尽的测试用例，以及在调试过程中不断学习和改进，可以开发出高效、稳定、用户友好的软件系统。未来，我将继续探索和学习新的技术，不断提升自己的软件设计和开发能力。