**Создание лавного потока main**

Этот код реализует многопоточный сервер на C++, который взаимодействует с динамической библиотекой (DLL) и использует API сокетов Windows (WinSock2) для обработки сетевых подключений. Также используются потоки для выполнения различных задач одновременно, таких как принятие клиентских подключений, распределение запросов, очистка ресурсов и обработка ввода/вывода через консоль.

**Глобальные переменные**

· **Обработчики потоков (**hAcceptServer**,** hDispatchServer**, и т.д.)**: Эти переменные представляют потоки, которые выполняют разные задачи.

· port**,** udpport: Порты для TCP и UDP по умолчанию.

· servercall: Идентификатор сервера, строковая переменная.

· **Критическая секция (**ListContactCriticalSection**)**: Используется для синхронизации доступа к общим ресурсам, в частности к списку контактов Contacts.

· sss: Указатель на функцию в загруженной DLL.

· st: Обработчик загруженной DLL.

· dllname**,** pipename: Пути к DLL и имя канала для коммуникации с консолью.

· CurrentClients**,** TotalClients**,** RejectedClients: Переменные для отслеживания состояния клиентов.

· AcceptEvent: Событие для обработки принятия соединений.

**Основная функция (main)**

1. **Обработка аргументов командной строки:**

Порты, путь к DLL и имя канала для консольного ввода-вывода задаются через аргументы командной строки. Если они не указаны, используются значения по умолчанию.

1. **Загрузка DLL:**

Путь к DLL преобразуется в широкие символы с помощью MultiByteToWideChar, и затем она загружается с помощью LoadLibrary.

Функция SSS загружается с помощью GetProcAddress.

1. **Инициализация сокетов (WinSock):**

Вызывается WSAStartup для инициализации библиотеки WinSock, необходимой для работы с сетевыми соединениями.

1. **Создание потоков:**

Создаются потоки для выполнения различных задач (прием подключений, обработка запросов, очистка ресурсов и т.д.).

Приоритеты потоков устанавливаются с помощью SetThreadPriority.

1. **Ожидание завершения потоков:**

Основной поток ожидает завершения всех созданных потоков с помощью WaitForSingleObject.

1. **Очистка ресурсов:**

После завершения работы потоков, дескрипторы потоков закрываются с помощью CloseHandle, освобождаются ресурсы и выполняется очистка WinSock с помощью WSACleanup.

**ConcurrentServer.h** Этот файл определяет структуру данных и интерфейсы для многопоточного сервера, который будет обрабатывать несколько клиентских подключений. Каждое подключение хранится в структуре **Contact**, которая содержит информацию о состоянии подключения и обслуживающего потока. Также определены функции для обработки команд сервера и обработки событий.

**AcceptServer**

Этот код представляет собой часть многопоточного сервера, использующего WinSock2 для обработки сетевых подключений и команд. Основная задача сервера — принимать подключенные клиенты и выполнять определенные команды, такие как старт, стоп, ожидание клиентов и завершение работы.

**AcceptCycle:**

Назначение: Эта функция принимает входящие подключения от клиентов.

В цикле она пытается принять подключение от клиента через сокет с использованием accept().

Если подключение не может быть принято (например, из-за того, что сокет находится в неблокирующем режиме и нет подключения), сервер ждет 100 миллисекунд.

Если клиент отключился или прервал соединение, выводится сообщение об этом.

Если подключение успешно, клиент добавляется в список подключений, и увеличивается счетчик текущих и общих клиентов.

В случае успешного приема подключения сервер продолжает ожидать других клиентов.

**CommandsCycle:**

Назначение: Обрабатывает команды, поступающие от управляющего потока.

Функция обрабатывает различные команды, такие как START (запуск сервера), STOP (остановка сервера), WAIT (ожидание всех клиентов) и SHUTDOWN (завершение работы).

Для команд START и STOP сервер выполняет соответствующие действия (запуск или остановка).

Для команды WAIT он ожидает завершения обслуживания всех клиентов, а затем снова переходит в состояние ожидания новых команд.

При команде SHUTDOWN сервер ждет завершения всех текущих подключений, после чего завершает работу.

**AcceptServer:**

Назначение: Основная функция, которая инициализирует серверный сокет и запускает цикл обработки команд.

Инициализирует библиотеку WinSock с помощью WSAStartup.

Создает серверный сокет для прослушивания входящих соединений.

Настроивает сокет для работы в неблокирующем режиме (используется функция ioctlsocket).

Привязывает сокет к указанному порту на всех интерфейсах (INADDR\_ANY).

Запускает процесс ожидания входящих соединений с помощью listen().

После этого передает управление функции CommandsCycle, где сервер начинает обрабатывать команды.

После завершения работы сервера закрывает сокет и очищает ресурсы с помощью WSACleanup.

**WaitClients:**

Назначение: Ожидание завершения обслуживания всех клиентов.

Что происходит:

Функция проверяет, все ли клиенты обслужены, используя критическую секцию для безопасного доступа к списку клиентов.

Если список клиентов пуст (все обслужены), сервер продолжает выполнение. Если нет, сервер ждет (с помощью SleepEx), пока все клиенты не будут обработаны.

**Принцип работы сервера:**

Когда сервер запускается, он инициализирует библиотеку WinSock и создает сокет.

Он настраивает серверный сокет в неблокирующем режиме и начинает слушать на порту.

Как только клиент подключается, сервер принимает подключение с помощью функции accept().

Принятые клиенты добавляются в список и обрабатываются.

Сервер продолжает принимать новых клиентов до тех пор, пока не получит команду завершения работы.

Основной поток управления обрабатывает различные команды:

* START — сервер начинает принимать клиентов.
* STOP — сервер останавливает обработку новых клиентов.
* WAIT — сервер ждет, пока все клиенты не будут обслужены.
* SHUTDOWN — завершение работы сервера после обслуживания всех клиентов.

**DispatchServer**

Функция **DispatchServer** представляет собой основной цикл сервера, который обрабатывает подключения клиентов, принимает сообщения от них, обрабатывает их и отправляет соответствующие ответы. Рассмотрим её поэтапно:

**1. Инициализация**

cout << "DispatchServer начал работу" << endl;char buffer[1024], Error[15] = "Ошибка запроса";int bytesReceived, bytesSent;

Вывод сообщения: сервер сообщает о начале работы функции.

Буфер для данных: buffer используется для хранения полученных данных от клиента.

Переменные для учета байтов: bytesReceived и bytesSent служат для хранения количества байтов, которые были получены или отправлены.

**2. Основной цикл**

while (\*((TalkersCommand\*)pPrm) != EXIT) {

Цикл обработки: сервер будет работать до тех пор, пока не получит команду EXIT. Команда EXIT передается через параметр pPrm (это указатель на объект типа TalkersCommand).

Ожидание события:

if (WaitForSingleObject(AcceptEvent, 100) == WAIT\_OBJECT\_0) {

Эта строка ожидает событие AcceptEvent, которое сигнализирует о наличии нового клиента для обработки.

Ожидание ограничено временем (100 миллисекунд), чтобы избежать блокировки в случае отсутствия новых клиентов.

**3. Работа с подключениями**

EnterCriticalSection(&ListContactCriticalSection);

Вход в критическую секцию: это нужно для безопасного доступа к общему списку подключений Contacts. Критическая секция защищает данные от одновременного изменения несколькими потоками.

for (auto it = Contacts.begin(); it != Contacts.end(); it++) {

if (it->type == Contact::ACCEPT) {

client = &(\*it);

Проход по списку контактов: цикл перебирает список клиентов (Contacts), чтобы найти тех, кто подключился (Contact::ACCEPT).

Проверка на подключение: если тип клиента — это ACCEPT, то мы выбираем этот контакт для дальнейшей работы.

**4. Получение данных от клиента**

bytesReceived = recv(client->s, buffer, sizeof(buffer) - 1, 0);

Получение данных: сервер пытается получить данные от клиента с помощью функции recv. Если данные успешно получены, они записываются в буфер buffer.

Проверка ошибок:

Если recv возвращает ошибку (например, клиент отключился или произошла другая ошибка), сервер закрывает соединение с клиентом.

if (bytesReceived == SOCKET\_ERROR) {

int err = WSAGetLastError();

if (err == WSAEWOULDBLOCK) {

Sleep(100);

} else if (err == WSAECONNABORTED || err == WSAECONNRESET || err == WSAENOTSOCK) {

closesocket(client->s);

client->s = INVALID\_SOCKET;

client->sthread = Contact::FINISH;

break;

} else {

cerr << SetErrorMsgText("Ошибка при приеме данных: ", GetLastError()) << endl;

break;

}

}

Обработка ошибок: если произошла ошибка при получении данных:

* Если ошибка — это неблокирующий режим (WSAEWOULDBLOCK), сервер ждет 100 миллисекунд.
* Если ошибка указывает на разрыв соединения (например, WSAECONNABORTED), сервер закрывает сокет и меняет состояние клиента на FINISH.
* В других случаях выводится сообщение об ошибке.

**5. Обработка полученных данных**

if (bytesReceived != SOCKET\_ERROR) {

buffer[bytesReceived] = '\0'; // Завершаем строку

cout << "\tСообщение от клиента: " << buffer << endl;

Если данные были получены успешно, они завершаются нулевым символом и выводятся в консоль.

client->type = Contact::CONTACT;

client->htimer = CreateWaitableTimer(NULL, false, NULL);

Установка типа клиента: теперь клиент считается активным (Contact::CONTACT).

Создание таймера: для клиента создается новый таймер (CreateWaitableTimer), который будет отсчитывать время. Это нужно, чтобы сервер мог завершить соединение с клиентом, если он не отвечает вовремя.

**6. Установка таймера**

\_int64 time = -300000000; // Таймер на 30 секундif (!SetWaitableTimer(client->htimer, (LARGE\_INTEGER\*)&time, 0, ASWTimer, client, false)) {

cerr << SetErrorMsgText("Не удалось установить таймер", GetLastError()) << endl;

CloseHandle(client->htimer);

client->htimer = NULL;

}

Таймер устанавливается на 30 секунд. Если клиент не отвечает вовремя, то сервер завершит его подключение.

**7. Создание потока для обработки клиента**

client->hthread = sss(buffer, client);if (!client->hthread) {

flag = false; // Поток не был создан

}

Если клиентский поток (hthread) не был создан, то флаг flag устанавливается в false.

**8. Ошибка или успешная обработка**

if (!flag) {

InterlockedIncrement(&RejectedClients);

if ((bytesSent = send(client->s, Error, sizeof(Error), NULL)) == SOCKET\_ERROR)

throw SetErrorMsgText("Ошибка при отправке данных: ", WSAGetLastError());

if (client->htimer != NULL) {

if (!CancelWaitableTimer(client->htimer)) {

cerr << SetErrorMsgText("Не удалось отменить таймер", GetLastError()) << endl;

CloseHandle(client->htimer);

client->htimer = NULL;

}

}

client->sthread = Contact::ABORT;

closesocket(client->s);

}

Если произошла ошибка (например, не удалось создать поток), сервер увеличивает счетчик отклоненных клиентов, отправляет ошибку клиенту и завершает соединение.

**9. Завершение цикла**

LeaveCriticalSection(&ListContactCriticalSection);

После обработки всех клиентов, мы выходим из критической секции, освобождая доступ к общим данным.

**10. Ожидание новых подключений**

SleepEx(0, TRUE); // Пауза

Система делает паузу и ждет новых событий.

**11. Завершение работы**

Когда сервер получает команду EXIT, цикл завершится, и функция завершит выполнение.

Функция DispatchServer отвечает за обработку всех подключений клиентов. Она принимает данные от клиентов, проверяет их, обрабатывает сообщения, создает потоки для обработки запросов и управляет таймерами. Если клиент не отвечает вовремя, соединение закрывается. Если все прошло успешно, сервер продолжает работать до тех пор, пока не получит команду на завершение.

**Реализация библиотеки обслуживающих серверов**

**DefineTableService** и макросы: В начале определены несколько серверных функций с использованием макросов:

BEGIN\_TABLESERVICE

ENTRYSERVICE("echo", EchoServer),

ENTRYSERVICE("time", TimeServer),

ENTRYSERVICE("rand", RandServer)

END\_TABLESERVICE;

Это выглядит как таблица, где каждому сервису присваивается уникальное имя и функция для обработки запроса.

**Функция SSS**: Это функция, экспортируемая как \_\_declspec(dllexport), которая принимает идентификатор сервиса (id) и параметры (prm), ищет соответствующую запись в таблице сервисов и создает новый поток для обработки сервиса:

HANDLE SSS(char\* id, LPVOID prm)

{

HANDLE rc = NULL;

int i = 0;

while (i < SIZETS && strcmp(TABLESERVICE\_ID(i), id) != 0) i++;

if (i < SIZETS) {

rc = CreateThread(NULL, NULL, TABLESERVICE\_FN(i), prm, NULL, NULL);

}

return rc;

};

Здесь SIZETS — размер таблицы, а TABLESERVICE\_ID(i) и TABLESERVICE\_FN(i) возвращают ID и функцию для сервиса.

**Серверы (EchoServer, RandServer, TimeServer)**: Каждая серверная функция работает с клиентом, который передается как параметр (Contact\* client). Все серверные функции делают следующее:

* Отправляют начальное сообщение клиенту.
* Принимают сообщения от клиента.
* Отправляют ответ обратно клиенту.
* Завершают работу при получении сигнала о завершении или ошибке.

**EchoServer (Эхо-сервер):** Этот сервер просто принимает сообщение от клиента и отправляет его обратно:

DWORD WINAPI EchoServer(LPVOID pPrm) {

// Инициализация и обработка сообщения

while (!client->TimerOff) {

if ((recv(client->s, ibuf, sizeof(ibuf), NULL)) == SOCKET\_ERROR) {

// Ошибка или таймаут

}

else {

send(client->s, ibuf, sizeof(ibuf), NULL); // Эхо

}

}

}

**RandServer (Сервер случайных чисел)**: Этот сервер генерирует случайное число и сравнивает его с числом, присланным клиентом:

DWORD WINAPI RandServer(LPVOID pPrm) {

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> distrib(1, 100);

int random\_number = distrib(gen);

// Обработка сообщений от клиента

}

**TimeServer (Сервер времени):** Этот сервер проверяет, совпадает ли время, присланное клиентом, с текущим временем на сервере:

DWORD WINAPI TimeServer(LPVOID pPrm) {

SYSTEMTIME stt;

GetLocalTime(&stt); // Текущее время на сервере

// Сравнение времени

}

**Использование APC (Asynchronous Procedure Calls):** Для вывода сообщений о начале и окончании работы серверов используется вызов QueueUserAPC, который асинхронно вызывает функции ASStartMessage и ASFinishMessage.

**Потоки и асинхронность:**

Каждая серверная функция запускается в отдельном потоке, что позволяет серверу обрабатывать несколько клиентов одновременно. С помощью CreateThread создаются новые потоки для обработки запросов каждого клиента.

**Реализация потока GarbageCleaner**

Функция GarbageCleaner в фоновом потоке периодически проверяет список клиентов. Если клиент завершил свою работу (состояние FINISH, TIMEOUT, или ABORT), то его соединение закрывается, а связанные ресурсы (сокет и поток) освобождаются. После этого клиент удаляется из списка, и уменьшается счетчик активных клиентов. Цикл повторяется каждую секунду, пока не получена команда завершения.

**Навигация сервера в локальной сети**

**1. Инициализация сокета и Winsock**

WSAStartup — инициализация библиотеки Winsock, которая необходима для работы с сетевыми сокетами на платформе Windows.

Если инициализация не удалась (возвращается ненулевой код ошибки), выбрасывается исключение с текстом ошибки.

**Создание сокета:**

socket(AF\_INET, SOCK\_DGRAM, NULL) создаёт сокет для работы с UDP. Если сокет не был создан (возвращается INVALID\_SOCKET), выбрасывается исключение с сообщением об ошибке.

**Установка неблокирующего режима:**

ioctlsocket(servsock, FIONBIO, &nonblk) — этот вызов устанавливает сокет в неблокирующий режим. Это значит, что операция чтения и записи с сокета не будет блокировать выполнение программы, если данные не готовы для передачи.

**Привязка сокета:**

bind(servsock, (LPSOCKADDR)&serv, sizeof(serv)) — связывает сокет с указанным адресом и портом. В данном случае это порт для UDP-сообщений. Если привязка не удается, выбрасывается исключение.

**2. Основной цикл сервера**

Цикл выполняется до тех пор, пока не поступит команда EXIT (выход из сервера).

**Прием данных:**

В цикле с помощью функции recvfrom сервер принимает данные от клиента. Эти данные записываются в буфер ibuf. Если данные не поступают, программа проверяет ошибку с помощью WSAGetLastError(). Если ошибка WSAEWOULDBLOCK (сокет временно не доступен), программа ждет 1 секунду перед повторной попыткой.

**Проверка и отправка данных:**

Если данные были получены (libuf > 0), сервер проверяет, совпадает ли содержимое буфера с заранее определенной строкой servercall.

Если строки совпадают, сервер отправляет полученные данные обратно клиенту с помощью функции sendto. Это сообщение отправляется через тот же сокет, от которого было получено.

**3. Завершение работы сервера**

После того как команда EXIT поступает в программу, цикл завершает свою работу.

**Закрытие сокета:**

В конце работы сокет закрывается с помощью closesocket(servsock). Если операция не удалась, выбрасывается исключение.

**Очистка ресурсов Winsock:**

После завершения работы вызывается WSACleanup(), чтобы корректно завершить использование библиотеки Winsock.

**4. Обработка ошибок**

Все ключевые операции в коде обрабатываются с использованием блока try-catch. Если возникает ошибка (например, проблемы с сокетом, с отправкой данных и т.д.), выбрасывается исключение с описанием ошибки. Эти исключения перехватываются в блоке catch, и сообщение об ошибке выводится в консоль.

**5. Вывод сообщений в консоль**

В начале и в конце работы сервера выводятся сообщения о запуске и остановке сервера на русском языке:

"Запуск ResponseServer..." — сообщает, что сервер начал свою работу.

"ResponseServer остановлен." — сообщает, что сервер завершил свою работу.

Важные моменты:

Неблокирующий режим: Этот режим полезен, чтобы сервер не блокировал выполнение программы в ожидании данных от клиента. В случае, если данные не приходят, сервер просто ждет некоторое время (в данном случае 1 секунду).

**Работа с UDP:** UDP — это протокол, не обеспечивающий гарантии доставки данных. Поэтому сервер в цикле получает данные и сразу их отправляет обратно, если они соответствуют предустановленному запросу (servercall).

**Обзор работы сервера:**

Этот сервер представляет собой многозадачное приложение на C++, которое управляет несколькими потоками для обслуживания различных задач, таких как принятие соединений, обработка данных, очистка ресурсов и взаимодействие с внешними библиотеками. Вот как функционирует сервер в целом:

**1. Обработка аргументов командной строки:**

При запуске программы можно указать параметры через командную строку, такие как:

* Порт для серверного соединения (port).
* Имя DLL-файла, который будет загружен для выполнения дополнительных операций.
* Порт для UDP-соединений (udpport).
* Строка, которая будет использоваться для проверки идентификации сервера (servercall).

Эти параметры обрабатываются функцией configureArguments, и если аргументы невалидны, сервер использует значения по умолчанию.

**2. Инициализация DLL:**

Сервер загружает внешнюю библиотеку (DLL), используя функцию initializeDll. Эта библиотека должна содержать функцию SSS, которая будет использоваться сервером для выполнения дополнительных операций.

**3. Инициализация WinSock:**

Для работы с сетевыми сокетами на Windows используется библиотека WinSock. Сервер инициализирует её с помощью функции initializeWinSock, чтобы начать работу с сокетами для TCP и UDP-соединений.

**4. Создание серверных потоков:**

Для выполнения различных задач сервер создаёт несколько потоков:

* AcceptServer — поток для принятия входящих клиентских соединений.
* DispatchServer — поток для обработки полученных данных и их распределения.
* GarbageCleaner — поток для очистки завершённых или неактивных соединений.
* ConsolePipe — поток для работы с каналом ввода/вывода в консоль.
* ResponseServer — поток для обработки запросов, связанных с UDP-соединениями (например, посылка ответов на запросы).

Потоки создаются с разными приоритетами (например, AcceptServer имеет самый высокий приоритет).

**5. Работа с критической секцией:**

Для синхронизации доступа к общим данным используется критическая секция ListContactCriticalSection, чтобы предотвратить гонки данных при работе с общими списками контактов (Contacts).

**6. Обработка соединений и команд:**

* Принятие соединений: Поток AcceptServer принимает входящие клиентские соединения. Для каждого клиента создаются новые потоки, которые обрабатывают их запросы.
* Обработка запросов: Поток DispatchServer управляет получением команд от клиентов и их распределением, возможно, в другие потоки для обработки.
* Очистка ресурсов: Поток GarbageCleaner периодически проверяет список контактов и удаляет завершённые или неактивные соединения, освобождая ресурсы.
* Ответы на запросы: Поток ResponseServer слушает UDP-порт и обрабатывает входящие сообщения от клиентов, отправляя ответы по заданному протоколу.

**7. Завершение работы:**

Когда сервер получает команду остановки (например, через консоль или по времени), он завершает работу.

Сервер ждёт завершения всех потоков с помощью WaitForSingleObject. После того как потоки завершили свою работу, сервер очищает все ресурсы (закрывает сокеты, очищает критическую секцию и завершает работу с WinSock).

**8. Обработка ошибок:**

Каждая важная операция (например, создание сокетов, принятие соединений, работа с DLL и т. д.) обрабатывается через исключения. В случае ошибки сервер выводит сообщение с кодом ошибки, чтобы информировать пользователя о возникшей проблеме.

**9. Взаимодействие с внешней DLL:**

Используемая DLL загружается динамически, и её функции вызываются сервером для выполнения определённых операций. В этом примере DLL используется для выполнения определённых сервисных задач, связанных с клиентскими запросами.

**10. Потоки с различными задачами:**

Каждый из потоков отвечает за свою задачу, что позволяет серверу обрабатывать несколько операций одновременно. Например, один поток может принимать новые соединения, а другой — обрабатывать запросы и отправлять ответы. Это позволяет улучшить производительность и не блокировать выполнение сервера при ожидании данных.

**Итог:**

Этот сервер организует асинхронную многозадачную обработку клиентов с использованием нескольких потоков для различных операций (принятие соединений, очистка ресурсов, обработка запросов). Сервер взаимодействует с внешними библиотеками через DLL и выполняет операции с помощью сетевых сокетов (TCP/UDP), обеспечивая высокую производительность и отказоустойчивость.

**EchoClient**

Данный код представляет собой реализацию UDP и TCP клиента, который обнаруживает сервер по широковещательной передаче (UDP), подключается к нему по TCP и отправляет/получает данные.

Функция GetServer

* Использует UDP широковещание для поиска сервера в локальной сети.
* Создает UDP-сокет.
* Настраивает широковещание через setsockopt с SO\_BROADCAST.
* Отправляет сообщение-запрос и ожидает ответ от сервера.
* Если ответ совпадает с запросом, сервер найден.

**Как это работает**

1. **Обнаружение сервера:**

Через UDP широковещание клиент ищет сервер, который отвечает на запрос.

1. **Подключение к серверу:**

После обнаружения серверного IP адреса клиент устанавливает TCP соединение.

1. **Передача данных:**

Клиент отправляет данные серверу и получает ответ, что имитирует эхо-сервер.

1. **Завершение:**

Сокет закрывается, освобождаются ресурсы WinSock.

**Основные особенности**

Использование UDP для поиска сервера.

Установка таймаутов и работа с IP через inet\_ntop и inet\_pton.

Логика обработки ошибок через пользовательские функции.

Последовательное использование UDP для обнаружения и TCP для передачи данных.

**Спецификация протокола взаимодействия клиентской консоли управления и сервера ConcurrentServer**

Клиентская консоль (RConsole) взаимодействует с сервером ConcurrentServer через именованные каналы (Named Pipes). Каждый клиентский запрос инициируется с помощью команды, которая отправляется на сервер через канал. Сервер выполняет соответствующую операцию и отправляет результат обратно клиенту. Взаимодействие между клиентом и сервером осуществляется через команды управления сервером, с возможностью получения диагностических сообщений.

**Принципы взаимодействия**

**- Использование именованных каналов:**

Клиентская консоль подключается к серверу через именованный канал, который определяется уникальным именем сервера.

Канал используется для отправки команд от клиента и получения ответов от сервера.

**- Командный интерфейс:**

Команды от клиента представляют собой числовые значения, отправляемые серверу через канал.

Сервер обрабатывает команды и возвращает соответствующие сообщения о статусе или результатах выполнения.

**- Формат сообщений:**

Команды клиента отправляются в виде строки с числовым значением.

Ответы сервера включают диагностические сообщения или результаты выполнения команды.

**- Типы команд:**

* start: Разрешает подключение клиентов к серверу.
* stop: Запрещает подключение клиентов к серверу.
* exit: Завершает работу сервера.
* statistics: Выводит статистику работы сервера.
* wait: Приостанавливает подключение клиентов до завершения работы с текущим подключением.
* shutdown: Выполняет последовательность команд: wait и exit.

**- API для клиента**

1. Подключение к серверу через именованный канал:

* HANDLE ConnectToServer(wchar\_t\* serverName)
* Параметры: имя сервера (именованный канал).
* Возвращает дескриптор канала (HANDLE), используемый для отправки команд.

1. **Отправка команды серверу:**

* bool SendCommand(HANDLE hNamedPipe, int command)
* Параметры: дескриптор канала, код команды.
* Возвращает true, если команда успешно отправлена.

1. **Чтение ответа от сервера:**

* string ReadResponse(HANDLE hNamedPipe)
* Параметры: дескриптор канала.
* Возвращает строку с ответом сервера.

1. **Закрытие соединения с сервером:**

* bool CloseConnection(HANDLE hNamedPipe)
* Параметры: дескриптор канала.
* Возвращает true, если соединение успешно закрыто.

**1. Критическая секция (Critical Section)**

Критическая секция — это механизм синхронизации, который используется для защиты определенной части кода от одновременного доступа нескольких потоков. В случае вашего сервера, критическая секция используется для защиты доступа к глобальному списку контактов (Contacts), чтобы избежать гонок потоков (race conditions), которые могут возникнуть, если несколько потоков одновременно изменяют или читают данные.

InitializeCriticalSection(&ListContactCriticalSection);EnterCriticalSection(&ListContactCriticalSection);// Работа с данными списка ContactsLeaveCriticalSection(&ListContactCriticalSection);DeleteCriticalSection(&ListContactCriticalSection);

В этом примере EnterCriticalSection блокирует доступ к критической секции другим потокам до тех пор, пока текущий поток не завершит работу в ней и не вызовет LeaveCriticalSection.

**2. Асинхронные процедуры**

Асинхронные процедуры в контексте вашего сервера, вероятно, означают работу с асинхронными сокетами или обработку событий без блокировки главного потока. Ваш сервер использует несколько потоков (например, AcceptServer, DispatchServer, GarbageCleaner и другие), каждый из которых выполняет свою задачу независимо, не блокируя другие операции.

Пример асинхронной работы:

* Поток AcceptServer занимается принятием входящих соединений, используя асинхронный метод для приема соединений.
* Другие потоки могут работать с данными клиентов, не блокируя друг друга.

Это позволяет серверу эффективно обрабатывать несколько запросов одновременно, минимизируя задержки.

**3. Как работает акцепт в не блокирующем режиме**

В не блокирующем режиме функция accept используется для принятия соединений, но она не блокирует выполнение программы, если нет подключений. Если соединение не установлено, то функция возвращает ошибку или выполняет другие действия, например, ожидает короткое время и проверяет снова.

u\_long mode = 1; // Включение неблокирующего режимаioctlsocket(serverSocket, FIONBIO, &mode);

SOCKET clientSocket = accept(serverSocket, NULL, NULL);if (clientSocket == INVALID\_SOCKET) {

// Если нет соединений, выполняем другие операции

}

В таком режиме сервер может продолжать выполнять другие задачи, не блокируясь в ожидании клиента. Это позволяет серверу быть более отзывчивым и масштабируемым.

**4. Механизм событий**

Механизм событий в Windows позволяет синхронизировать потоки, сигнализируя им, когда нужно выполнить определенную задачу. События могут быть установлены или сброшены, и потоки могут ожидать их изменения.

В вашем сервере используется событие AcceptEvent, которое позволяет потоку принимать соединения или выполнять другие действия, когда оно установлено.

AcceptEvent = CreateEvent(NULL, FALSE, FALSE, NULL);// Ждем, пока событие не будет установленоWaitForSingleObject(AcceptEvent, INFINITE);

Поток может ожидать установки события другим потоком, после чего будет продолжена обработка.

**5. Таймер**

Таймеры в Windows используются для выполнения действий через заданные интервалы времени. В вашем сервере используется таймер, чтобы ограничить время работы обслуживающих потоков или выполнить периодические задачи (например, очистку ресурсов).

SetTimer(hWnd, TIMER\_ID, 1000, (TIMERPROC)ASWTimer); // Таймер с интервалом 1 секунда

Таймер может быть использован для выполнения повторяющихся задач или для завершения операции, если она выполняется слишком долго.

**6. Атомарные операции**

Атомарные операции — это операции, которые выполняются как единое целое, без возможности прерывания другими потоками. Это важно для работы с переменными, которые могут быть изменены несколькими потоками одновременно, например, счетчиками клиентов.

В вашем сервере переменные CurrentClients, TotalClients, и RejectedClients помечены как volatile LONG для атомарных операций, что предотвращает гонки потоков при изменении этих переменных.

InterlockedIncrement(&CurrentClients); // Атомарное увеличение счетчика

Использование InterlockedIncrement или других функций атомарных операций помогает избежать ошибок, связанных с одновременным доступом.

**7. Загрузка библиотеки**

В вашем сервере используется динамическая загрузка библиотеки с помощью функции LoadLibrary для загрузки DLL (например, ServiceLibrary.dll), и последующий доступ к функциям этой библиотеки через GetProcAddress.

st = LoadLibrary(wDllName);if (!st) {

cerr << "Ошибка при загрузке DLL: " << GetLastError() << endl;

return 1;

}

sss = (HANDLE(\*)(char\*, LPVOID))GetProcAddress(st, "SSS");if (!sss) {

cerr << "Ошибка при получении адреса функции: " << GetLastError() << endl;

FreeLibrary(st);

return 1;

}

Это позволяет серверу загружать и использовать внешние библиотеки без необходимости перекомпиляции сервера. Такая гибкость полезна для расширяемости и использования сторонних сервисов.