**Критическая секция**

Критическая секция — это механизм синхронизации, обеспечивающий защиту участка кода, который может быть одновременно использован несколькими потоками. Она предотвращает гонки потоков (race conditions), которые возникают, если несколько потоков одновременно читают или изменяют общий ресурс.

На вашем сервере критическая секция используется для синхронизации доступа к общему списку контактов (Contacts). Это позволяет гарантировать корректность работы с данными, когда:

Несколько потоков одновременно обращаются к списку.

Операции записи или чтения выполняются без конфликтов.

Основные переменные и ресурсы

ListContactCriticalSection:

Механизм синхронизации для защиты списка контактов.

Управляет доступом потоков к объекту Contacts.

sss: Указатель на функцию в загруженной DLL.

st: Обработчик загруженной DLL.

dllname: Путь к динамической библиотеке (DLL).

pipename: Имя канала для коммуникации между сервером и консолью.

CurrentClients: Текущее количество подключенных клиентов.

TotalClients: Общее количество обработанных подключений.

RejectedClients: Количество отклоненных подключений.

Пример работы с критической секцией

// Инициализация критической секции

InitializeCriticalSection(&ListContactCriticalSection);

// Вход в критическую секцию

EnterCriticalSection(&ListContactCriticalSection);

// Работа с общими данными (например, список контактов)

Contacts.Add(newContact); // Пример операции с ресурсом

// Выход из критической секции

LeaveCriticalSection(&ListContactCriticalSection);

// Удаление критической секции после завершения работы

DeleteCriticalSection(&ListContactCriticalSection);

Пошаговый процесс

InitializeCriticalSection(&ListContactCriticalSection) — создает объект критической секции перед началом работы.

EnterCriticalSection(&ListContactCriticalSection) — поток блокирует доступ другим потокам к защищенному коду.

Код внутри критической секции выполняется только одним потоком. Это может быть:

Добавление нового контакта.

Чтение данных.

Удаление контакта.

LeaveCriticalSection(&ListContactCriticalSection) — поток освобождает доступ для других потоков.

DeleteCriticalSection(&ListContactCriticalSection) — очищает ресурсы, связанные с критической секцией.

**Асинхронные процедуры:**

Асинхронные процедуры — это механизмы выполнения задач, которые не блокируют основной поток программы. Они позволяют серверу обрабатывать несколько операций одновременно, повышая производительность и минимизируя задержки.

В вашем сервере асинхронность реализована через Потоки для обработки клиентов.

APC (Asynchronous Procedure Calls) для выполнения функций в асинхронном режиме.

Использование APC (Asynchronous Procedure Calls)

APC позволяет асинхронно вызывать функции в контексте потока, который находится в состоянии ожидания.

Вызов функции:

Для обработки сообщений о начале и окончании работы серверов используется функция QueueUserAPC. Она добавляет вызовы функций ASStartMessage и ASFinishMessage в очередь APC потока.

Пример работы:

QueueUserAPC((PAPCFUNC)ASStartMessage, threadHandle, (ULONG\_PTR)startData);

QueueUserAPC((PAPCFUNC)ASFinishMessage, threadHandle, (ULONG\_PTR)finishData);

ASStartMessage: Отправляет сообщение о начале работы сервера.

ASFinishMessage: Отправляет сообщение об окончании работы сервера.

threadHandle: Указывает поток, в котором будет выполнен APC.

startData / finishData: Данные, передаваемые в функцию.

Потоки и асинхронность

Каждая серверная функция запускается в отдельном потоке, что обеспечивает независимость задач и позволяет:

* Принимать новые подключения.
* Обрабатывать запросы клиентов параллельно.
* Осуществлять очистку или другие фоновые задачи.
* Создание потоков: С помощью CreateThread создаются потоки для обработки запросов клиентов.

Ваш сервер реализует многозадачность через несколько потоков, каждый из которых выполняет определенную задачу. Это позволяет эффективно обрабатывать запросы клиентов.

Пример распределения задач:

AcceptServer: Принимает входящие соединения. Использует асинхронный метод для работы с подключениями.

DispatchServer: Отвечает за маршрутизацию данных между клиентами и сервером.

GarbageCleaner: Очищает ресурсы, освобождая память и закрывая неиспользуемые соединения.

**Работа функции accept в неблокирующем режиме**

В неблокирующем режиме функция accept используется для принятия новых соединений от клиентов. В отличие от блокирующего режима, где выполнение программы приостанавливается до момента установки соединения, в неблокирующем режиме accept сразу возвращает управление программе.

Если на момент вызова accept нет доступных подключений: Функция возвращает INVALID\_SOCKET.

Программа может продолжить выполнение других задач и позже повторно проверить наличие новых соединений.

Настройка неблокирующего режима

Для перевода сокета в неблокирующий режим используется функция ioctlsocket:

u\_long mode = 1; // 1 включает неблокирующий режим, 0 — блокирующий

ioctlsocket(serverSocket, FIONBIO, &mode);

serverSocket: Сокет сервера, который принимает соединения.

FIONBIO: Флаг для установки неблокирующего режима.

mode: Переменная, управляющая режимом работы сокета.

После установки этого режима вызовы функций ввода-вывода, таких как accept, не будут блокировать выполнение программы.

Пример кода

u\_long mode = 1;

ioctlsocket(serverSocket, FIONBIO, &mode); // Переводим сокет в неблокирующий режим

while (true) {

SOCKET clientSocket = accept(serverSocket, NULL, NULL);

if (clientSocket == INVALID\_SOCKET) {

// Если соединения нет, выполняем другие операции

printf("No connections available, continuing other tasks...\n");

Sleep(100); // Ожидаем перед повторной проверкой

} else {

// Успешно принято соединение

printf("Client connected!\n");

HandleClient(clientSocket); // Обработка клиента

}

}

Особенности работы в неблокирующем режиме

Если accept возвращает INVALID\_SOCKET, вызов функции WSAGetLastError() позволяет определить причину ошибки.

Код ошибки WSAEWOULDBLOCK указывает, что в данный момент нет подключений, но сокет работает корректно.

Параллельные задачи:

Сервер может выполнять другие операции, например:

* Обработку данных уже подключенных клиентов.
* Очистку ресурсов.
* Подготовку к новым запросам.
* Проверка через таймеры: Важно регулярно проверять наличие новых подключений, например, с помощью цикла с короткими паузами (Sleep).

**Механизм событий: использование и работа**

Механизм событий в Windows предоставляет средства для синхронизации потоков. Событие используется для уведомления одного или нескольких потоков о том, что определенное состояние достигнуто или задача выполнена.

Типы событий:

* Автоматически сбрасываемые события (Auto-reset events): После того как событие установлено, оно автоматически сбрасывается, и только один поток продолжает выполнение.
* Ручное управление событиями (Manual-reset events): Событие остается установленным до тех пор, пока его явно не сбросят.

На вашем сервере событие AcceptEvent используется для уведомления потока о необходимости принять новое соединение.

Создание события:

HANDLE AcceptEvent = CreateEvent(

NULL, // Атрибуты безопасности

FALSE, // Автоматический сброс события

FALSE, // Начальное состояние — не установлено

NULL // Имя события

);

NULL: Указывает, что событие локальное и не может быть доступно другим процессам.

FALSE (автоматический сброс): После обработки событие автоматически сбрасывается.

FALSE (начальное состояние): Указывает, что событие изначально не установлено.

Ожидание события

Функция WaitForSingleObject: Используется для ожидания установки события.

Пример ожидания с таймаутом:

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

// Событие установлено — принимаем соединение

printf("New connection detected.\n");

AcceptConnection(); // Обработка нового соединения

} else {

// Таймаут — выполняем другие задачи

printf("No connections, continuing other operations.\n");

}

AcceptEvent: Событие, которое поток ожидает.

100: Таймаут в миллисекундах. Указывает максимальное время ожидания.

WAIT\_OBJECT\_0: Возвращается, если событие установлено.

Пример ожидания без ограничения по времени:

WaitForSingleObject(AcceptEvent, INFINITE);

// После установки события продолжается выполнение

AcceptConnection();

Установка события: Событие устанавливается для уведомления ожидающих потоков.

SetEvent(AcceptEvent);

Сброс события (для ручного управления): Сбрасывает событие, чтобы оно стало неактивным.

ResetEvent(AcceptEvent);

Пример работы с событием

#include <windows.h>

#include <stdio.h>

HANDLE AcceptEvent;

DWORD WINAPI ClientHandler(LPVOID param) {

while (1) {

// Ожидание установки события

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

printf("Processing new client...\n");

// Обработка соединения

Sleep(500); // Имитация обработки

}

}

return 0;

}

int main() {

// Создание события

AcceptEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

// Создание потока для обработки клиентов

HANDLE clientThread = CreateThread(NULL, 0, ClientHandler, NULL, 0, NULL);

// Установка события для симуляции нового подключения

while (1) {

printf("Simulating new connection...\n");

SetEvent(AcceptEvent); // Уведомление потока о новом соединении

Sleep(2000); // Имитация интервала между соединениями

}

// Закрытие событий и потоков (обычно в конце работы программы)

CloseHandle(AcceptEvent);

CloseHandle(clientThread);

return 0;

}

**Использование таймеров в Windows для управления соединениями**

Таймеры в Windows используются для выполнения действий через заданные интервалы времени. Они помогают управлять временем выполнения операций и ограничивать длительность задач, что особенно полезно для серверов при работе с клиентскими соединениями.

В вашем сервере таймер используется для:

Ограничения времени ожидания ответа от клиента.

Выполнения повторяющихся задач, таких как очистка ресурсов.

Создание таймера

Для создания таймера применяется функция CreateWaitableTimer:

HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);

NULL: Указывает, что используются стандартные атрибуты безопасности.

TRUE: Таймер является ручным (manual-reset), то есть не сбрасывается автоматически.

NULL: Таймер создается без имени, доступен только текущему процессу.

Таймер устанавливается с помощью функции SetWaitableTimer, которая задает время срабатывания и (опционально) периодичность выполнения:

\_\_int64 time = -300000000; // Таймер на 30 секунд (в 100-наносекундных интервалах)

if (!SetWaitableTimer(hTimer, (LARGE\_INTEGER\*)&time, 0, TimerCallback, client, FALSE)) {

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

CloseHandle(hTimer);

}

hTimer: Дескриптор таймера, созданного через CreateWaitableTimer.

(LARGE\_INTEGER\*)&time: Указатель на время, заданное в 100-наносекундных интервалах. Отрицательное значение означает время до истечения таймера (отсчет назад).

0: Указывает, что таймер не повторяется.

TimerCallback: Указатель на функцию обратного вызова, которая выполняется при срабатывании таймера.

client: Пользовательские данные, передаваемые в функцию обратного вызова.

FALSE: Указывает, что таймер не асинхронный.

Когда таймер срабатывает, вызывается функция обратного вызова, например:

void CALLBACK TimerCallback(LPVOID lpArg, DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

Client\* client = (Client\*)lpArg;

// Завершение соединения с клиентом

printf("Client timeout. Closing connection...\n");

CloseConnection(client);

}

lpArg: Указатель на данные клиента (например, структуру Client), переданные в SetWaitableTimer.

dwTimerLowValue и dwTimerHighValue: Значения таймера, которые можно использовать для синхронизации.

Пример кода

#include <windows.h>

#include <iostream>

using namespace std;

struct Client {

HANDLE hTimer;

SOCKET socket;

};

void CALLBACK TimerCallback(LPVOID lpArg, DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

Client\* client = (Client\*)lpArg;

cout << "Timeout reached. Closing client connection..." << endl;

// Закрытие соединения

closesocket(client->socket);

CloseHandle(client->hTimer);

}

int main() {

Client client;

// Создание таймера

client.hTimer = CreateWaitableTimer(NULL, TRUE, NULL);

if (client.hTimer == NULL) {

cerr << "Не удалось создать таймер: " << GetLastError() << endl;

return 1;

}

// Установка таймера на 30 секунд

\_\_int64 time = -300000000; // 30 секунд

if (!SetWaitableTimer(client.hTimer, (LARGE\_INTEGER\*)&time, 0, TimerCallback, &client, FALSE)) {

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

CloseHandle(client.hTimer);

return 1;

}

// Основной цикл

cout << "Waiting for timer..." << endl;

WaitForSingleObject(client.hTimer, INFINITE);

return 0;

}

Для выполнения периодических задач, таких как проверка состояния клиентов или очистка ресурсов, используется повторяющийся таймер. Для этого в параметре Period функции SetWaitableTimer указывается интервал повторения:

SetWaitableTimer(hTimer, (LARGE\_INTEGER\*)&time, 1000, TimerCallback, NULL, FALSE);

1000: Интервал в миллисекундах между срабатываниями.

Для приложений с оконным интерфейсом используется функция SetTimer, которая вызывает указанную функцию через заданные интервалы:

SetTimer(hWnd, TIMER\_ID, 1000, (TIMERPROC)TimerProc);

hWnd: Дескриптор окна, связанного с таймером.

TIMER\_ID: Идентификатор таймера.

1000: Интервал в миллисекундах.

TimerProc: Функция, вызываемая при срабатывании таймера.

**Атомарные операции**

Атомарные операции — это операции, которые выполняются целиком и не могут быть прерваны другими потоками. Они гарантируют целостность данных при работе с переменными, доступ к которым осуществляется одновременно из нескольких потоков.

В многопоточном приложении, например, сервере, переменные, такие как счетчики клиентов (активных, отклоненных или всего обработанных), могут быть одновременно изменены несколькими потоками. Это может привести к состоянию гонки (race condition) и некорректным результатам.

На сервере для работы с переменными CurrentClients, TotalClients, и RejectedClients применяются атомарные операции. Эти переменные объявлены как volatile LONG, что обеспечивает:

Компилятор или процессор не будет кэшировать значения переменных, а всегда будет обращаться к их актуальному значению в памяти.

Операции над переменными выполняются без прерываний, обеспечивая их корректность.

Пример:

volatile LONG CurrentClients = 0;

volatile LONG TotalClients = 0;

volatile LONG RejectedClients = 0;

Функции для атомарных операций

Windows API предоставляет функции для работы с атомарными операциями. Основные из них:

InterlockedIncrement — Увеличивает значение переменной на 1.

InterlockedDecrement — Уменьшает значение переменной на 1.

InterlockedExchange — Заменяет значение переменной на новое.

InterlockedCompareExchange — Выполняет сравнение и замену значения.

Пример: Увеличение счетчика клиентов

InterlockedIncrement(&CurrentClients); // Атомарное увеличение счетчика активных клиентов

&CurrentClients: Указатель на переменную, значение которой увеличивается.

Пример: Уменьшение счетчика клиентов

Когда клиент отключается, счетчик активных клиентов уменьшается:

InterlockedDecrement(&CurrentClients); // Атомарное уменьшение счетчика активных клиентов

Пример: Проверка и замена значения

Для более сложных сценариев используется InterlockedCompareExchange, чтобы проверить текущее значение и заменить его, если оно соответствует ожидаемому:

LONG expected = 5;

LONG newValue = 10;

LONG result = InterlockedCompareExchange(&CurrentClients, newValue, expected);

// Если CurrentClients == 5, то заменится на 10. В противном случае ничего не изменится.

&CurrentClients: Указатель на переменную.

newValue: Новое значение, которое будет установлено.

expected: Ожидаемое значение переменной.

**Загрузка библиотеки (DLL) в Windows**

Динамическая загрузка библиотек позволяет приложению загружать и использовать функции из внешних библиотек (DLL) во время выполнения. Это обеспечивает гибкость, позволяя заменять или обновлять библиотеки без перекомпиляции основного приложения.

Загрузка библиотеки

Используется функция LoadLibrary, которая загружает указанную DLL в адресное пространство процесса. Если библиотека загружена успешно, возвращается дескриптор (handle).

Получение адреса функции

С помощью GetProcAddress можно получить адрес экспортируемой функции из загруженной DLL.

Освобождение библиотеки

После завершения работы с библиотекой используется FreeLibrary, чтобы освободить ресурсы.

Пример кода

#include <windows.h>

#include <iostream>

using namespace std;

int main() {

const char\* dllName = "ServiceLibrary.dll"; // Имя библиотеки

const char\* functionName = "SSS"; // Имя функции

// Загрузка библиотеки

HMODULE st = LoadLibrary(dllName);

if (!st) {

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

return 1;

}

cout << "DLL загружена успешно." << endl;

// Получение адреса функции

typedef HANDLE(\*SSSFunction)(char\*, LPVOID); // Объявление типа функции

SSSFunction sss = (SSSFunction)GetProcAddress(st, functionName);

if (!sss) {

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

FreeLibrary(st);

return 1;

}

cout << "Функция " << functionName << " загружена успешно." << endl;

// Использование функции

// Например, вызов sss("параметры", nullptr);

// Освобождение библиотеки

FreeLibrary(st);

cout << "DLL освобождена." << endl;

return 0;

}

Детали работы

Загрузка библиотеки:

HMODULE st = LoadLibrary("ServiceLibrary.dll");

if (!st) {

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

}

LoadLibrary: Загружает библиотеку.

GetLastError: Возвращает код ошибки, если загрузка не удалась.

Получение адреса функции:

SSSFunction sss = (SSSFunction)GetProcAddress(st, "SSS");

if (!sss) {

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

FreeLibrary(st);

}

GetProcAddress: Возвращает указатель на функцию, экспортируемую DLL.

Освобождение библиотеки:

FreeLibrary(st);

FreeLibrary: Освобождает загруженную библиотеку, возвращая системе используемые ресурсы.

Обработка пути к библиотеке

Если путь к библиотеке задается в виде строки в кодировке ANSI, его можно преобразовать в широкую строку с помощью MultiByteToWideChar, чтобы поддерживать работу с Unicode:

WCHAR wDllName[MAX\_PATH];

MultiByteToWideChar(CP\_ACP, 0, dllName, -1, wDllName, MAX\_PATH);

HMODULE st = LoadLibrary(wDllName);