**Министерство цифрового развития, связи и массовых**

**коммуникаций Российской Федерации**

**Ордена Трудового Красного Знамени**

**федеральное государственное бюджетное**

**учреждение высшего образования**

**«Московский Технический Университет Связи и Информатики»**

Кафедра «Системное программирование»

**КУРСОВАЯ РАБОТА**

по теме «Разработка системы мониторинга

системных ресурсов вычислительного устройства

под управлением операционной системы Windows»

Выполнил: студент группы БСТ2001

Савкин Д. И.

Проверил: асс. каф. СП

Алексанян Д. А.

Москва 2022

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ 3

ОСНОВНАЯ ЧАСТЬ 4

1.1 Краткие теоретические основы 4

1.1.1 Почтовые ящики 5

1.1.2 Каналы 5

1.1.3 Разделяемая память 6

1.1.4 Мьютекс 7

1.1.5 Событие 7

1.1.6 Семафор 7

1.1.7 Ждущие таймеры 7

1.1.8 Сокет 8

1.2 Разработка программного продукта 8

1.2.1 Разработка протокола сетевого взаимодействия 9

1.2.2 Разработка клиентского приложения 11

1.2.3 Разработка серверного приложения 14

ЗАКЛЮЧЕНИЕ 16

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 17

ПРИЛОЖЕНИЕ 18

# ВВЕДЕНИЕ

**Актуальность темы.** Автоматизация процессов мониторинга системных параметров стала неотъемлемой частью бизнес-процесса любой информационной системы. Зачастую физический доступ к ЭВМ в рамках рабочего процесса невозможен в виду фактора уровня доступа сотрудника, недостижимый удалённости самого объекта и т.д. В результате появилась необходимость получать информацию с удалённого устройства в целях контроля общего состояния и поиска причин возникших ошибок в производственной работе.

**Целью** данной курсовой работы является закрепление и углубление теоретических знаний в области современных операционных систем, приобретение практических навыков разработки клиент-серверных приложений, использующих стандартные механизмы межпроцессного взаимодействия.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. изучить методы разработки клиент-серверных приложений;
2. изучить механизмы обмена данными в Windows NT и особенности их применения на практике;
3. изучить средства синхронизации в Windows NT и особенности из применения на практике;
4. изучить функции Win32 API для получения системной информации, информации о процессах и потоках, состояния памяти;
5. разработать серверные приложения и приложение-клиент, обменивающиеся данными через заданный интерфейс;
6. оформить результаты выполнения курсовой работы в виде пояснительной записки в соответствии с предъявленными требованиями.

**Заданием** на курсовую работу является разработка многозадачного приложения, выполняющего получение, сбор и вывод системной информации: количество жёстких дисков с типом их файловой системы (сервер 1) и объём физической и свободной оперативной памяти в указанных клиентом единицах (сервер 2).

# ОСНОВНАЯ ЧАСТЬ

## Краткие теоретические основы

Разрабатываемая система основывается на дуплексном взаимодействии двух приложений, запущенных на ЭВМ клиента и сервера соответственно. Взаимодействие двух приложений происходит в рамках вычислительной архитектуры «клиент-сервер». Такая схема обмена данными выбрана ввиду следующих достоинств:

1. Делает возможным, в большинстве случаев, распределение функций вычислительной системы между несколькими независимыми компьютерами в сети. Это позволяет упростить обслуживание вычислительной системы. В частности, замена, ремонт, модернизация или перемещение сервера не затрагивают клиентов.
2. Все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов. На сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа.
3. Позволяет объединить различные клиенты. Использовать ресурсы одного сервера часто могут клиенты с разными аппаратными платформами, операционными системами и т. п.

К недостаткам такого подхода можно отнести:

1. Неработоспособность сервера может сделать неработоспособной всю вычислительную сеть.
2. Поддержка работы данной системы требует отдельного специалиста — системного администратора.
3. Высокая стоимость оборудования.

Рассмотренная архитектура определяет лишь принцип взаимодействия приложений между собой. Реализация же определяется поддерживаемым набором системных средств коммуникации и синхронизации процессов.

Межпроцессорное взаимодействие предполагает взаимодействие между параллельными процессами как в рамках одной системы, так и в рамках нескольких систем, работающих на разных вычислительных системах. Рассмотрим некоторые методы межпроцессорного взаимодействия в Windows NT.

### Почтовые ящики

Почтовые ящики обеспечивают только однонаправленные соединения. Каждый процесс, который создаёт почтовый ящик, является «сервером почтовых ящиков». Другие процессы, называемые «клиентами почтовых ящиков», посылают сообщения серверу, записывая их в почтовый ящик. Входящие сообщения всегда дописываются в почтовый ящик и сохраняются до тех пор, пока сервер их не прочтёт. Каждый процесс может одновременно быть и сервером и клиентом почтовых ящиков, создавая, таким образом, двунаправленные коммуникации между процессами.

Клиент может посылать сообщения в почтовый ящик, расположенный на том же компьютере, на компьютере в сети, или на все почтовые ящики с одним именем всем компьютерам выбранного домена. При этом широковещательное сообщение, транслируемое по домену, не может быть более 400 байт. В остальных случаях размер сообщения ограничивается только при создании почтового ящика сервером.

Почтовый ящик является псевдофайлом, находящимся в памяти. Для того, чтобы получить к нему доступ, необходимо использовать стандартные функции для работы с файлами. Данные в почтовом ящике могут быть в любой форме – их интерпретирует прикладная программа, но их общий объем не должен превышать 64 Кб. В отличие от дисковых файлов, почтовые ящики являются временными — когда все дескрипторы почтового ящика закрыты, он и все его данные удаляются. Все почтовые ящики являются локальными по отношению к создавшему их процессу; процесс не может создать удалённый mailslot.

### Каналы

Существует два способа организовать двунаправленное соединение с помощью каналов: безымянные и именованные каналы.

Безымянные (или анонимные) каналы позволяют связанным процессам передавать информацию друг другу. Обычно, безымянные каналы используются для перенаправления стандартного ввода/вывода дочернего процесса так, чтобы он мог обмениваться данными с родительским процессом. Чтобы производить обмен данными в обоих направлениях, необходимо создать два безымянных канала. Родительский процесс записывает данные в первый канал, используя его дескриптор записи, в то время как дочерний процесс считывает данные из канала, используя дескриптор чтения. Аналогично, дочерний процесс записывает данные во второй канал и родительский процесс считывает из него данные. Безымянные каналы не могут быть использованы для передачи данных по сети и для обмена между несвязанными процессами.

Именованные каналы используются для передачи данных между независимыми процессами или между процессами, работающими на разных компьютерах. Обычно, процесс сервера именованных каналов создаёт именованный канал с известным именем или с именем, которое будет передано клиентам. Процесс клиента именованных каналов, зная имя созданного канала, открывает его на своей стороне с учётом ограничений, указанных процессом сервера. После этого между сервером и клиентом создаётся соединение, по которому может производиться обмен данными в обоих направлениях.

### Разделяемая память

Разделяемая память является самым быстрым средством обмена данными между процессами. При использовании других средств межпроцессного взаимодействия обмен информацией между процессами происходит через ядро, что приводит к переключению контекста между процессом и ядром и, следовательно, к потерям производительности.

Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов. Сегмент разделяемой памяти подключается к свободной части виртуального адресного пространства процесса. Таким образом, в двух разных процессах могут быть разные адреса одной и той же ячейки, подключённой разделяемой памяти. О физической памяти говорят, что она совместно используется, если она отображается на виртуальное адресное пространство нескольких процессов.

Принципиальная схема такого метода показана на рисунке 1.

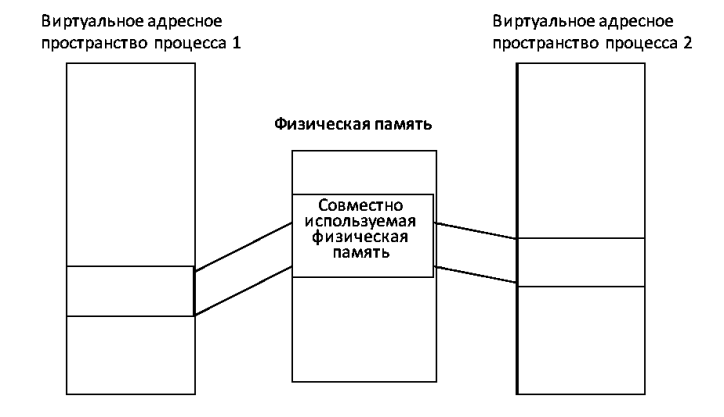


Рисунок — Принципиальная схема метода разделения памяти

### Мьютекс

Мьютекс — это объект ядра, который можно использовать для синхронизации потоков разных процессов. Он может принадлежать или не принадлежать некоторому потоку. Если мьютекс принадлежит потоку, то он находится в состоянии «занято». Если данный объект не относится ни к одному потоку, то он находится в состоянии «свободно». Другими словами, принадлежать для него означает быть в состоянии «занято». Никакой другой поток не может «завладеть» мьютексом, если тот находится в состоянии «занято». Таким образом, механизм очень похож на действие механизма критической секции.

### Событие

События используются в качестве сигналов о завершении какой-либо операции. Однако в отличие от мьютексов, они не принадлежат ни одному потоку. Например, поток А создает событие с помощью функции и устанавливает объект в состояние «занято». Поток В получает дескриптор этого объекта, затем вызывает системную функцию, чтобы приостановить работу до того момента, когда поток А завершит конкретную задачу и освободит указанный объект. Когда это произойдёт, система выведет из состояния ожидания поток В, который теперь владеет информацией, что поток А завершил выполнение своей задачи.

### Семафор

В семафорах применён принцип действия мьютексов, но с добавлением одной существенной детали. В них заложена возможность подсчёта ресурсов, что позволяет заранее определённому числу потоков одновременно войти в синхронизуемый участок кода. Семафоры могут быть полезны при совместном использовании ограниченных ресурсов. Предположим, имеется три приложения, каждое из которых должно выполнить вывод на печать, а у компьютера только два параллельных порта. Установив семафор с начальным значением счётчика ресурсов, равным двум, можно заставить приложения запрашивать сервис печати только тогда, когда есть свободный параллельный порт.

### Ждущие таймеры

Ждущий таймер — это также объект ядра операционной системы. Он работает в трёх режимах. В режиме «ручного сброса» таймер переходит в установленное состояние при истечении заданной задержки и остаётся установленным до тех пор, пока не будет задана новая задержка. В режиме «автоматического сброса» таймер переходит в установленное состояние при истечении заданной задержки и остаётся установленным до первого успешного вызова функции ожидания. В этом режиме он напоминает объект Событие в режиме автоматического сброса, поскольку каждый раз при истечении времени задержки разрешается выполнение лишь одной нити. Наконец, ждущий таймер может выполнять функции интервального таймера, который перезапускается с заданной задержкой после каждого срабатывания объекта.

Главная особенность, отличающая ждущие таймеры от системных, — то, что ждущие таймеры могут совместно использоваться несколькими приложениями. Например, можно приостановить несколько приложений в фоновом режиме так, чтобы они «просыпались» каждые несколько часов для выполнения некоторой операции.

Процессы получают дескрипторы ждущих таймеров так же, как они получают дескрипторы мьютексов: дублированием, наследованием или открытием по имени.

### Сокет

Сокет — механизм операционной системы, позволяющий устанавливать соединения с другими устройствами, зная их сетевой адрес и номер порта. Обычно различают клиентские и серверные сокеты. Сокеты клиента можно сравнить с аппаратом телефонной сети, когда как серверные сокеты больше походят на коммутаторы. Каждый процесс может создать как клиентский, так и серверный сокет. Для создания последнего требуется выполнить «привязку» к какому-либо свободному и разрешённому систему номеру порта. Это необходимо для того, чтоб клиентский сокет, указав тот же самый номер порта и адрес серверного сокета, мог «достучаться» и, при необходимости, присоединиться к нему (в случае TCP-соединения). Слушающий процесс (сокет) обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения.

Рассмотренные методы межпроцессного взаимодействия по-своему реализуют задачу передачи данных. В данной работе процесс обмена сообщениями между клиентом и сервером имплементирован на основе последнего варианта — сокетов — ввиду необходимости использования кроссплатформенного механизма (в случае будущего расширения системы) для передачи информации в рамках сетевой организации подключения устройств.

## Разработка программного продукта

Для разработки программного продукта были определены следующие подзадачи:

1. определить функционал каждого приложения системы;
2. разработать универсальный протокол взаимодействия клиента и сервера;
3. установить последовательности передачи/обработки/формирования/передачи данных.

В качестве инструментария был выбран высокоуровневый язык программирования ANSI C89 в имплементации от компании Microsoft.

Клиент-серверное взаимодействие, как было сказано ранее, позволяет распределить функций вычислительной системы между несколькими независимыми компьютерами в сети. Это обосновано тем, что серверные устройства обладают большей мощностью и объёмом обрабатываемого информационного ресурса, чем устройства клиента. Ввиду этого факта, были определены следующие задачи для клиентского приложения:

1. реализовать интерфейс, позволяющий пользователю подключиться по выбору к любому серверу;
2. реализовать возможность раздельного получения и отображения данных для каждой командой, определяемой протоколом передачи данных.

Серверные приложения представляют собой отдельные процессы, запускаемые на том же или другом устройстве относительно клиента. Для них были определены следующие задачи:

1. получение количества жёстких дисков и тип файловой системы каждого (сервер 1);
2. получение количества логических процессоров (сервер 1);
3. объём физической памяти в единицах, переданных клиентом (в байтах, мегабайтах, гигабайтах по выбору пользователя) (сервер 2);
4. объём свободной физической памяти в единицах, переданных клиентом (в байтах, мегабайтах, гигабайтах по выбору пользователя) (сервер 2).

### Разработка протокола сетевого взаимодействия

Протокол сетевого взаимодействия определяет правила передачи сообщений между устройствами. В совокупности можно выделить следующие типы взаимодействия между устройствами: одностороннее и двустороннее.

Одностороннее взаимодействие предполагает, что инициирующая сторона соединения формирует и отправляет одиночные сообщения другой стороне, которая формирует и отправляет одиночное сообщение в ответ. В такой схеме только инициирующая сторона является первоначальным источником сообщений, тогда как принимающая лишь «отвечает» на поступающие запросы.

Двустороннее взаимодействие предполагает, что каждая сторона одновременно является как инициирующей, так и принимающей. В такой схеме порядок отправки и получения сообщений регулируется набором правил в целях обеспечения целостности и доступности данных.

Протокол, реализованный в рамках текущей работы, придерживается модели двустороннего взаимодействия по следующим причинам:

1. двусторонний тип передачи сообщений позволяет более гибко и в сравнительно быстрые сроки расширять протокол и добавлять новый функционал без возникновения проблем с поддержкой предыдущей версией протокола;
2. двусторонний тип передачи упрощает модификацию существующих правил обмена сообщениями.

Сообщения, передающиеся в рамках протокола, сериализуются (переводятся в битовую последовательность) и отправляются по каналу связи по инициативе любой из сторон, тогда как принимающая сторона получает данную последовательность и десериализует (переводит в структуру данных).

Каждое сообщение, передаваемое клиентом, имеет два параметра: вид и тип. Значение параметра вида определяет схему взаимодействия клиента и сервера, тогда как значение параметра типа — информацию о запрашиваемых данных.

Протокол поддерживает несколько видов сообщений:

1. Unary (унарный).  
   Унарные сообщения интерпретируются серверов как одиночный запрос и требуют от него [сервера] одиночного ответа. Следующее сообщение не будет отправлено клиентом до тех пор, пока не будет получен ответ на предыдущее сообщение.
2. Timer (интервальный).  
   Интервальные сообщения интерпретируются сервером так же, как и унарные, за исключением того, что сервер не отправляет сообщения с запрашиваемыми данными, если те не поменялись с момента получения предыдущего сообщения. Такие сообщения отправляются клиентом по таймеру. Интервальные сообщения содержат специальное поле, являющимся маркером актуальности данных. Значение маркера в сообщении сервера интерпретирует клиентом как некоторое целое неотрицательное число, поставленное в соответствие с моментом времени, во время которого было сформировано сообщение. Значение маркера в сообщении клиента интерпретируется сервером как некоторое целое неотрицательное число, обозначающее фактическое присутствие и осведомлённость клиента о сообщении, сформированным в тот момент времени, в соответствии с которым поставлено значение этого числа.
3. Реактивный (reactive).  
   Реактивные сообщения интерпретируются сервером как «подписка» клиента на сообщения об изменении информации со стороны сервера. Реактивная модель позволяет в режиме реального времени получать информацию от сервера без необходимости запрашивать её вновь.

В рамках текущей работы протокол поддерживает следующие типы сообщений: hdinfo, cpuinfo, flram и avram. Спецификация протокола гарантирует поддержку унарного вида для каждого типа сообщений, однако поддержка остальных видов для каждого отдельного типа определяется реализацией конкретного сервера.

Сообщения типа hdinfo позволяют получить информацию о количестве жёстких дисков, установленных на сервере, и тип файловой системы каждого.

Сообщение типа cpuinfo позволяют получить информацию о количестве логических процессоров, установленных на сервере.

Сообщение типа flram позволяет получить информацию об объёме физической памяти в единицах, переданных клиентом (в байтах, мегабайтах, гигабайтах по выбору пользователя). Параметр единицы измерения может принимать следующие значения: b, kb, mb, gb, что означает байты, кибибайты, мебибайты и гибибайты соответственно.

Сообщение типа avram позволяет получить информацию об объёме свободной физической памяти в единицах, переданных клиентом (в байтах, мегабайтах, гигабайтах по выбору пользователя). Значения параметра единиц измерения аналогичны.

### Разработка клиентского приложения

Клиентское приложение должно реализовывать интерфейс, позволяющий пользователю подключаться по выбору к любому серверу. Интерфейс любого приложения можно разделить на два типа: консольные и графические. Консольные приложения запускаются пользователем из-под командной строки и используют стандартные потоки ввода/вывода для взаимодействия с внешней (по отношению к самому процессу) средой. Графические приложения используют средства операционной системы для создания т.н. окон и заполняют визуальный контейнер графическими элементами: текст, кнопки, списки, изображения и т.д. Основная идеология создания приложений в операционной системе Windows — программирование графических приложений. Специалисты из Microsoft предлагают программистам использовать широкий функционал актуальных прикладных интерфейсов (API) для разработки конечных продуктов. Однако не всегда именно графическое представление удачно вписывается в архитектуру разрабатываемого программного комплекса. Средства администрирования и другие продукты индустриального качества зачастую реализуют два подхода одновременно: позволяют работать как из-под командной строки, так и путём графического взаимодействия с пользователем.

Разрабатываемое клиентское приложение имеет довольно небольшой функционал, поэтому было решено реализовать его в виде консольного приложения, соблюдая следующие условия:

1. приложение имеет чёткую и понятную инструкцию пользования;
2. приложение должно запускаться на персональных компьютерах, имеющих низкий уровень вычислительных ресурсов, и обладать высоким уровнем быстродействия.
3. приложение должно в чёткой и понятной для пользователя форме выводит результаты работы вычислительных процессов, а также информировать пользователя о возникающих ошибках как со стороны самого клиента, так и со стороны сервера, с которым производится взаимодействие.

Формат входных данных клиентского приложения представлен в следующем виде:

<server> <port> <kind> <command> [<param>]..., где

<server> — IP-адрес сервера, к которому будет производиться подключение;

<port> — сетевой порт сервера, к которому будет производиться подключение;

<kind> — вид подключения, определяющий способ передачи/получения информации;

<command> — команда для выполнения;

<param> — опциональный (необязательный) набор параметров для указанной команды.

Допустимые значения аргументов <kind>, <command> и <param> определяются спецификацией протокола клиент-серверного взаимодействия.

Пояснительное сообщение (инструкция) выводится в стандартный поток вывода ошибок и имеет вид, показанный на рисунке 2.

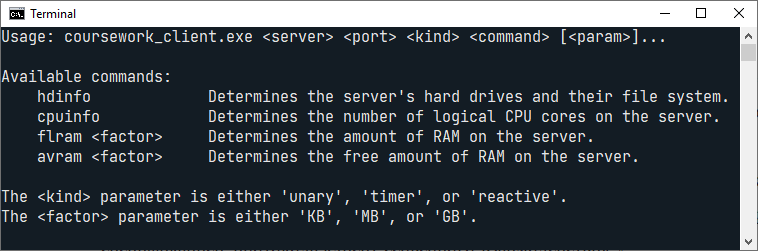


Рисунок — Реализация пояснительного сообщения клиентского приложения

Пояснительное сообщение выводится каждый раз, когда переданный набор входных данных не соответствует установленному шаблону.

Сообщения об ошибках выводятся непосредственно в момент её возникновения в следующем формате:

ERROR: <message>, где

<message> — расширенное сообщение об ошибке.

Пример сообщения об ошибке показан на рисунке 3.

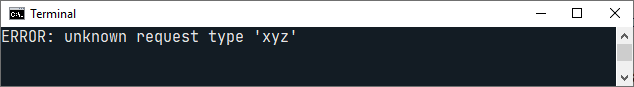


Рисунок — Пример сообщения об ошибке на клиенте

В процессе работы клиентское приложение записывает полученные результаты в стандартный поток вывода. Пример результата запроса типа hdinfo в унарном, интервальном и реактивном виде показан на рисунке 4.

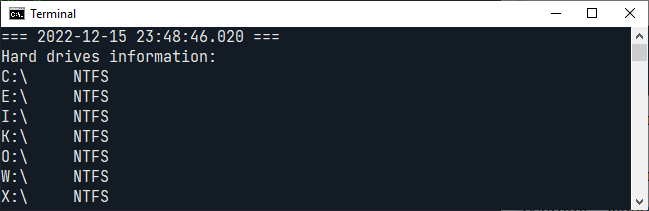


Рисунок — Пример результат запроса информации о дисках и их файловых системах

### Разработка серверного приложения

Серверное приложение должно реализовывать многопоточную обработку клиентов, то есть допускать одновременное подключение нескольких клиентов. Серверное приложение обрабатывает сообщения клиента в соответствии с указанными параметрами вида и типа и формирует ответное сообщение (при необходимости), дополнительно передавая текущее время в качестве значения соответствующего поля ответного сообщения. Формат входных данных серверного приложения представлен в следующем виде: <port>, где <port> — сетевой порт, который будет выступать в качестве «слушающего» порта для организации подключения клиентов.

Пояснительное сообщение (инструкция) выводится в стандартный поток вывода ошибок и имеет вид, показанный на рисунке 5.

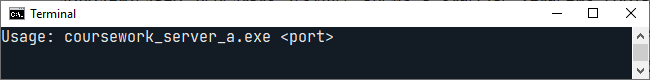


Рисунок — Реализация пояснительного сообщения серверного приложения

Пояснительное сообщение выводится каждый раз, когда переданный набор входных данных не соответствует установленному шаблону.

Сообщения об ошибках сервера идентичны по формату с сообщениями об ошибках клиента. Пример сообщения об ошибке показан на рисунке 6.

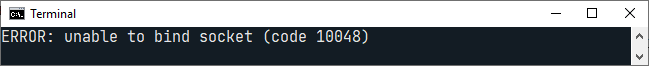


Рисунок — Пример сообщения об ошибке на сервере

В процессе обновления информации сервер записывает уведомление о прошедшем обновлении в стандартный поток вывода. Это показано на рисунке 7.



Рисунок — Уведомление сервера об обновлении информации

# ЗАКЛЮЧЕНИЕ

В рамках работы над курсовым проектом была разработана системы мониторинга системных ресурсов вычислительного устройства под управлением операционной системы Windows.

В результате выполнения работы были решены следующие задачи:

1. изучены методы разработки клиент-серверных приложений;
2. изучены механизмы обмена данными в Windows NT и особенности их применения на практике;
3. изучены средства синхронизации в Windows NT и особенности из применения на практике;
4. изучены функции Win32 API для получения системной информации, информации о процессах и потоках, состояния памяти;
5. разработаны серверные приложения и приложение-клиент, обменивающиеся данными через заданный интерфейс;
6. оформлены результаты выполнения курсовой работы в виде пояснительной записки в соответствии с предъявленными требованиями.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 7.32-2017. Межгосударственный стандарт. Система стандартов по информации, библиотечному и издательскому делу. Отчёт о научно-исследовательской работе. Структура и правила оформления (введён в действие Приказом Росстандарта от 24.10.2017 N 1494-ст) – URL: https://docs.cntd.ru/document/1200157208 (дата обращения: 15.12.2022). – Текст: электронный.
2. Windows API index. — URL: https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list (дата обращения 15.12.2022).

# ПРИЛОЖЕНИЕ

Исходный код клиентского приложения:

#define WIN32\_LEAN\_AND\_MEAN

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <assert.h>

#include <stdarg.h>

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#include "../../common/protocol.h"

#pragma comment(lib, "ws2\_32.lib")

#define fatal\_error(fmt, ...) do { error(fmt, \_\_VA\_ARGS\_\_); exit(1); } while (0)

enum {

BUFFER\_SIZE = 512,

};

static void error(const char\* const fmt, ...) {

va\_list args;

va\_start(args, fmt);

printf("ERROR: ");

vprintf(fmt, args);

printf("\n");

va\_end(args);

}

static void print\_timestamp(SYSTEMTIME time) {

printf("=== %04hu-%02hu-%02hu %02hu:%02hu:%02hu.%03hu ===\n",

time.wYear, time.wMonth, time.wDay,

time.wHour, time.wMinute, time.wSecond, time.wMilliseconds);

}

static Request create\_request(char\* kind, char\* type) {

Request request = { 0 };

for (RequestKind it = REQ\_NONE; it < REQ\_COUNT; it++) {

if (kind\_names[it] && !strcmp(kind, kind\_names[it])) {

request.kind = it;

break;

}

}

for (RequestType it = TYPE\_NONE; it < TYPE\_COUNT; it++) {

if (type\_names[it] && !strcmp(type, type\_names[it])) {

request.type = it;

break;

}

}

if (!request.kind) {

fatal\_error("unknown request kind '%s'", kind);

}

if (!request.type) {

fatal\_error("unknown request type '%s'", type);

}

return request;

}

static Response parse\_response(char\* buf) {

Response response = \*(Response\*)buf;

print\_timestamp(response.time);

if (response.error) {

switch (response.error) {

case ERROR\_EMPTY\_RESPONSE: break;

case ERROR\_UNSUPPORTED\_REQUEST\_KIND: {

error("request kind is not supported by server");

goto end;

}

case ERROR\_UNSUPPORTED\_REQUEST\_TYPE: {

error("request type is not supported by server");

goto end;

}

default: {

if (response.error < 0) {

error("server responded with error code %d", -response.error);

} else if (response.error != ERROR\_EMPTY\_RESPONSE) {

error("unknown response error code %d", response.error);

}

goto end;

} break;

}

}

switch (response.type) {

case TYPE\_HARD\_DRIVE\_INFO: {

printf("Hard drives information:");

if (response.hard\_drive\_info.count) {

printf("\n");

for (uint8\_t i = 0; i < response.hard\_drive\_info.count; i++) {

DriveInfo\* info = response.hard\_drive\_info.infos + i;

printf("%c:\\\t%s\n", info->letter, info->fs);

}

} else {

printf(" NONE\n");

}

} break;

case TYPE\_CPU\_INFO: {

printf("Server logical CPU cores: %lu\n", response.logical\_cpu\_count);

} break;

case TYPE\_FULL\_RAM: {

printf("Server total RAM: %llu %s\n", response.ram\_info.value, factor\_names[response.ram\_info.factor]);

} break;

case TYPE\_AVAILABLE\_RAM: {

printf("Server available RAM: %llu %s\n", response.ram\_info.value, factor\_names[response.ram\_info.factor]);

} break;

default: error("unknown response type %d", response.type); break;

}

end:

printf("\n");

return response;

}

static uint32\_t factorize(char\* factor) {

for (char\*\* name = factor\_names;

name != factor\_names + sizeof(factor\_names) / sizeof(\*factor\_names);

name++

) {

if (\*name && !\_stricmp(\*name, factor)) {

return name - factor\_names;

}

}

error("unknown factor parameter '%s', using bytes by default", factor);

return 0;

}

static void print\_usage(void) {

fprintf(stderr,

"Usage: coursework\_client.exe <server> <port> <kind> <command> [<param>]...\n\n"

"Available commands:\n"

" hdinfo Determines the server's hard drives and their file system.\n"

" cpuinfo Determines the number of logical CPU cores on the server.\n"

" flram <factor> Determines the amount of RAM on the server.\n"

" avram <factor> Determines the free amount of RAM on the server.\n\n"

"The <kind> parameter is either 'unary', 'timer', or 'reactive'.\n"

"The <factor> parameter is either 'KB', 'MB', or 'GB'.\n");

exit(1);

}

int main(int argc, char\* argv[]) {

argc--, argv++;

if (argc < 4) {

print\_usage();

}

char\* server = argv[0];

char\* port = argv[1];

Request request = create\_request(argv[2], argv[3]);

if ((request.type == TYPE\_FULL\_RAM) ||

(request.type == TYPE\_AVAILABLE\_RAM)) {

if (argc < 5) {

print\_usage();

}

request.factor = factorize(argv[4]);

}

WSADATA data;

int result = WSAStartup(MAKEWORD(2, 2), &data);

if (result) {

fatal\_error("unable to start WSA (code %d)", result);

}

struct addrinfo hints = { 0 };

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

struct addrinfo\* info;

result = getaddrinfo(server, port, &hints, &info);

if (result) {

error("unable to get address info (code %d)", result);

WSACleanup();

return 1;

}

SOCKET sock = socket(info->ai\_family, info->ai\_socktype, info->ai\_protocol);

if (sock == INVALID\_SOCKET) {

error("unable to create socket (code %d)", WSAGetLastError());

WSACleanup();

return 1;

}

result = connect(sock, info->ai\_addr, (int)info->ai\_addrlen);

if (result == SOCKET\_ERROR) {

closesocket(sock);

sock = INVALID\_SOCKET;

}

freeaddrinfo(info);

if (sock == INVALID\_SOCKET) {

error("unable to connect to %s:%s (code %d)", server, port, WSAGetLastError());

WSACleanup();

return 1;

}

char buf[BUFFER\_SIZE] = { 0 };

switch (request.kind) {

case REQ\_UNARY: {

result = send(sock, (char\*)&request, sizeof(Request), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

break;

}

result = recv(sock, buf, sizeof(buf), 0);

if (result > 0) {

parse\_response(buf);

} else if (result < 0) {

error("unable to receive bytes (code %d)", WSAGetLastError());

}

} break;

case REQ\_TIMER: {

do {

result = send(sock, (char\*)&request, sizeof(Request), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

break;

}

result = recv(sock, buf, sizeof(buf), 0);

if (result > 0) {

Response response = parse\_response(buf);

if (response.error != ERROR\_EMPTY\_RESPONSE) {

request.ts = response.ts;

}

} else if (result < 0) {

error("unable to receive bytes (code %d)", WSAGetLastError());

}

Sleep(3000);

} while (result > 0);

} break;

case REQ\_REACTIVE: {

result = send(sock, (char\*)&request, sizeof(Request), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

break;

}

do {

result = recv(sock, buf, sizeof(buf), 0);

if (result > 0) {

parse\_response(buf);

} else if (result < 0) {

error("unable to receive bytes (code %d)", WSAGetLastError());

}

} while (result > 0);

} break;

default: assert(request.kind == REQ\_NONE); break;

}

result = shutdown(sock, SD\_BOTH);

if (result == SOCKET\_ERROR) {

error("unable to shutdown socket (code %d)", WSAGetLastError());

closesocket(sock);

WSACleanup();

return 1;

}

closesocket(sock);

WSACleanup();

return 0;

}

Исходный код серверного приложения для сервера 1:

#define WIN32\_LEAN\_AND\_MEAN

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <assert.h>

#include <stdarg.h>

#include <stdbool.h>

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#include "../../common/protocol.h"

#pragma comment(lib, "ws2\_32.lib")

#define fatal\_error(fmt, ...) do { error(fmt, \_\_VA\_ARGS\_\_); exit(1); } while (0)

enum {

BUFFER\_SIZE = 512,

};

static void error(const char\* const fmt, ...) {

va\_list args;

va\_start(args, fmt);

printf("ERROR: ");

vprintf(fmt, args);

printf("\n");

va\_end(args);

}

static void\* xmalloc(size\_t size) {

void\* ptr = malloc(size);

if (!ptr) {

fatal\_error("unable to allocate memory");

}

return ptr;

}

static DWORD global\_drive\_last\_error;

static uint32\_t global\_drive\_ts;

static uint8\_t global\_drive\_count;

static DriveInfo global\_drive\_infos[26];

static CRITICAL\_SECTION global\_drive\_critical\_section;

static CONDITION\_VARIABLE global\_drive\_condition\_variable;

static DWORD get\_drive\_info(uint32\_t in\_ts, uint32\_t\* out\_ts, DriveInfo\* out\_infos, uint8\_t\* out\_count, bool force) {

EnterCriticalSection(&global\_drive\_critical\_section);

while ((in\_ts >= global\_drive\_ts) && !force) {

SleepConditionVariableCS(&global\_drive\_condition\_variable, &global\_drive\_critical\_section, INFINITE);

}

\*out\_ts = global\_drive\_ts;

memcpy(out\_infos, global\_drive\_infos, sizeof(DriveInfo) \* global\_drive\_count);

\*out\_count = global\_drive\_count;

DWORD error = global\_drive\_last\_error;

LeaveCriticalSection(&global\_drive\_critical\_section);

return error;

}

static void update\_drive\_info(void) {

char drives[BUFFER\_SIZE] = { 0 };

char\* p = drives;

if (!GetLogicalDriveStringsA(sizeof(drives) - 1, drives)) {

DWORD code = GetLastError();

error("unable to get logical drive strings (code %lu)", code);

return;

}

EnterCriticalSection(&global\_drive\_critical\_section);

uint8\_t count = 0;

DriveInfo tmp[sizeof(global\_drive\_infos) / sizeof(DriveInfo)] = { 0 };

DriveInfo\* entry = tmp;

while (\*p) {

if (!GetVolumeInformationA(p, NULL, 0, NULL, NULL, NULL, entry->fs, sizeof(entry->fs) - 1)) {

DWORD code = GetLastError();

if (code != ERROR\_NOT\_READY) { // nocheckin

error("unable to get volume '%c' information (code %lu)", \*p, code);

global\_drive\_last\_error = code;

}

goto skip;

}

count++;

entry->letter = \*p;

entry++;

skip:

while (\*p++);

}

bool should\_wakeup = false;

if (memcmp(global\_drive\_infos, tmp, sizeof(DriveInfo) \* count)) {

global\_drive\_ts++;

memcpy(global\_drive\_infos, tmp, sizeof(DriveInfo) \* count);

global\_drive\_count = count;

should\_wakeup = true;

printf("Drive information updated\n");

}

LeaveCriticalSection(&global\_drive\_critical\_section);

if (should\_wakeup) {

WakeAllConditionVariable(&global\_drive\_condition\_variable);

}

}

static DWORD count\_bits(ULONG\_PTR mask) {

DWORD shift = sizeof(ULONG\_PTR) \* 8 - 1;

DWORD count = 0;

ULONG\_PTR test = (ULONG\_PTR)1 << shift;

for (DWORD i = 0; i <= shift; i++) {

count += ((mask & test) ? 1 : 0);

test /= 2;

}

return count;

}

static DWORD get\_cpu\_info(unsigned long\* out\_logical\_cpu\_count) {

PSYSTEM\_LOGICAL\_PROCESSOR\_INFORMATION info = NULL;

DWORD len = 0, offset = 0, count = 0;

retry:

if (!GetLogicalProcessorInformation(info, &len)) {

DWORD code = GetLastError();

if (code == ERROR\_INSUFFICIENT\_BUFFER) {

info = xmalloc(len);

goto retry;

}

error("unable to get logical processor information (code %lu)", code);

free(info);

return code;

}

PSYSTEM\_LOGICAL\_PROCESSOR\_INFORMATION it = info;

while (offset + sizeof(SYSTEM\_LOGICAL\_PROCESSOR\_INFORMATION) <= len) {

if (it->Relationship == RelationProcessorCore) {

count += count\_bits(it->ProcessorMask);

}

offset += sizeof(SYSTEM\_LOGICAL\_PROCESSOR\_INFORMATION);

it++;

}

\*out\_logical\_cpu\_count = count;

return ERROR\_SUCCESS;

}

static DWORD WINAPI process\_client(LPVOID param) {

SOCKET client = (SOCKET)param;

retry:

char buf[BUFFER\_SIZE] = { 0 };

int result = recv(client, buf, sizeof(buf), 0);

if (result < 0) {

error("unable to receive bytes (code %d)", WSAGetLastError());

closesocket(client);

return 1;

} else if (result > 0) {

Request request = \*(Request\*)buf;

Response response = { .type = request.type };

switch (request.type) {

case TYPE\_HARD\_DRIVE\_INFO: {

switch (request.kind) {

case REQ\_UNARY: {

response.error = 0 - get\_drive\_info(

0, &response.ts,

response.hard\_drive\_info.infos,

&response.hard\_drive\_info.count,

false

);

} break;

case REQ\_TIMER: {

response.error = 0 - get\_drive\_info(

request.ts,

&response.ts,

response.hard\_drive\_info.infos,

&response.hard\_drive\_info.count,

true

);

if (request.ts >= response.ts) {

Response empty = {

.type = response.type,

.error = ERROR\_EMPTY\_RESPONSE

};

response = empty;

}

GetSystemTime(&response.time);

result = send(client, (char\*)&response, sizeof(Response), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

closesocket(client);

return 1;

}

goto retry;

} break;

case REQ\_REACTIVE: {

uint32\_t ts = 0;

for (;;) {

response.error = 0 - get\_drive\_info(

ts, &ts,

response.hard\_drive\_info.infos,

&response.hard\_drive\_info.count,

false

);

GetSystemTime(&response.time);

result = send(client, (char\*)&response, sizeof(Response), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

closesocket(client);

return 1;

}

}

} break;

default: {

error("unsupported request kind %d", request.kind);

response.error = ERROR\_UNSUPPORTED\_REQUEST\_KIND;

} break;

}

} break;

case TYPE\_CPU\_INFO: {

switch (request.kind) {

case REQ\_UNARY: {

response.error = 0 - get\_cpu\_info(&response.logical\_cpu\_count);

} break;

default: {

error("unsupported request kind %d", request.kind);

response.error = ERROR\_UNSUPPORTED\_REQUEST\_KIND;

} break;

}

} break;

default: {

error("unknown request type %d", request.type);

response.error = ERROR\_UNSUPPORTED\_REQUEST\_TYPE;

} break;

}

GetSystemTime(&response.time);

result = send(client, (char\*)&response, sizeof(Response), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

}

}

result = shutdown(client, SD\_BOTH);

if (result == SOCKET\_ERROR) {

error("unable to shutdown client (code %d)", WSAGetLastError());

closesocket(client);

return 1;

}

return 0;

}

static DWORD WINAPI update\_drive\_info\_routine(LPVOID param) {

(void)param;

for (;;) {

update\_drive\_info();

Sleep(3000);

}

}

int main(int argc, char\* argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: coursework\_server\_a.exe <port>\n");

exit(1);

}

InitializeCriticalSection(&global\_drive\_critical\_section);

InitializeConditionVariable(&global\_drive\_condition\_variable);

WSADATA data;

int result = WSAStartup(MAKEWORD(2, 2), &data);

if (result) {

fatal\_error("unable to start WSA (code %d)", result);

}

struct addrinfo hints = { 0 };

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

hints.ai\_flags = AI\_PASSIVE;

struct addrinfo\* info;

result = getaddrinfo(NULL, argv[1], &hints, &info);

if (result) {

error("unable to get address info (code %d)", result);

WSACleanup();

return 1;

}

SOCKET sock = socket(info->ai\_family, info->ai\_socktype, info->ai\_protocol);

if (sock == INVALID\_SOCKET) {

error("unable to create socket (code %d)", WSAGetLastError());

freeaddrinfo(info);

WSACleanup();

return 1;

}

result = bind(sock, info->ai\_addr, (int)info->ai\_addrlen);

if (result == SOCKET\_ERROR) {

error("unable to bind socket (code %d)", WSAGetLastError());

freeaddrinfo(info);

closesocket(sock);

WSACleanup();

return 1;

}

freeaddrinfo(info);

result = listen(sock, SOMAXCONN);

if (result == SOCKET\_ERROR) {

error("unable to listen socket (code %d)", WSAGetLastError());

closesocket(sock);

WSACleanup();

return 1;

}

HANDLE thread = CreateThread(NULL, 0, update\_drive\_info\_routine, NULL, 0, NULL);

if (thread == INVALID\_HANDLE\_VALUE) {

error("unable to create routine thread (code %d)", WSAGetLastError());

closesocket(sock);

WSACleanup();

return 1;

}

for (;;) {

SOCKET client = accept(sock, NULL, NULL);

if (client == INVALID\_SOCKET) {

error("unable to accept client (code %d)", WSAGetLastError());

closesocket(client);

continue;

}

thread = CreateThread(NULL, 0, process\_client, (LPVOID)client, 0, NULL);

if (thread == INVALID\_HANDLE\_VALUE) {

error("unable to create thread for client (code %lu)", GetLastError());

}

}

}

Исходный код серверного приложения для сервера 2:

#define WIN32\_LEAN\_AND\_MEAN

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <assert.h>

#include <stdarg.h>

#include <stdbool.h>

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#include "../../common/protocol.h"

#pragma comment(lib, "ws2\_32.lib")

#define fatal\_error(fmt, ...) do { error(fmt, \_\_VA\_ARGS\_\_); exit(1); } while (0)

enum {

BUFFER\_SIZE = 512,

};

static void error(const char\* const fmt, ...) {

va\_list args;

va\_start(args, fmt);

printf("ERROR: ");

vprintf(fmt, args);

printf("\n");

va\_end(args);

}

static DWORD global\_ram\_last\_error;

static uint32\_t global\_ram\_ts;

static DWORDLONG global\_total\_ram;

static DWORDLONG global\_available\_ram;

static CRITICAL\_SECTION global\_ram\_critical\_section;

static CONDITION\_VARIABLE global\_ram\_condition\_variable;

static DWORD get\_ram\_info(uint32\_t in\_ts, uint32\_t\* out\_ts, DWORDLONG\* out\_total\_ram, DWORDLONG\* out\_available\_ram, bool force) {

EnterCriticalSection(&global\_ram\_critical\_section);

while ((in\_ts >= global\_ram\_ts) && !force) {

SleepConditionVariableCS(&global\_ram\_condition\_variable, &global\_ram\_critical\_section, INFINITE);

}

if (out\_total\_ram) {

\*out\_total\_ram = global\_total\_ram;

}

if (out\_available\_ram) {

\*out\_available\_ram = global\_available\_ram;

}

\*out\_ts = global\_ram\_ts;

DWORD error = global\_ram\_last\_error;

LeaveCriticalSection(&global\_ram\_critical\_section);

return error;

}

static void update\_ram\_info(void) {

MEMORYSTATUSEX status = { sizeof(status) };

if (!GlobalMemoryStatusEx(&status)) {

DWORD code = GetLastError();

error("unable to get global memory status (code %lu)", code);

return;

}

EnterCriticalSection(&global\_ram\_critical\_section);

bool should\_wakeup = false;

if ((global\_total\_ram != status.ullTotalPhys) ||

(global\_available\_ram != status.ullAvailPhys)) {

global\_total\_ram = status.ullTotalPhys;

global\_available\_ram = status.ullAvailPhys;

global\_ram\_ts++;

should\_wakeup = true;

printf("Memory information updated\n");

}

LeaveCriticalSection(&global\_ram\_critical\_section);

if (should\_wakeup) {

WakeAllConditionVariable(&global\_ram\_condition\_variable);

}

}

static DWORD WINAPI process\_client(LPVOID param) {

SOCKET client = (SOCKET)param;

retry:

char buf[BUFFER\_SIZE] = { 0 };

int result = recv(client, buf, sizeof(buf), 0);

if (result < 0) {

error("unable to receive bytes (code %d)", WSAGetLastError());

closesocket(client);

return 1;

} else if (result > 0) {

Request request = \*(Request\*)buf;

Response response = { .type = request.type };

switch (request.type) {

case TYPE\_FULL\_RAM: {

switch (request.kind) {

case REQ\_UNARY: {

DWORDLONG value = 0;

response.error = 0 - get\_ram\_info(0, &response.ts, &value, NULL, false);

value /= 1ULL << request.factor;

response.ram\_info.factor = request.factor;

response.ram\_info.value = value;

} break;

default: {

error("unsupported request kind %d", request.kind);

response.error = ERROR\_UNSUPPORTED\_REQUEST\_KIND;

} break;

}

} break;

case TYPE\_AVAILABLE\_RAM: {

switch (request.kind) {

case REQ\_UNARY: {

DWORDLONG value = 0;

response.error = 0 - get\_ram\_info(0, &response.ts, NULL, &value, false);

value /= 1ULL << request.factor;

response.ram\_info.factor = request.factor;

response.ram\_info.value = value;

} break;

case REQ\_TIMER: {

DWORDLONG value = 0;

response.error = 0 - get\_ram\_info(0, &response.ts, NULL, &value, true);

value /= 1ULL << request.factor;

response.ram\_info.factor = request.factor;

response.ram\_info.value = value;

if (request.ts >= response.ts) {

Response empty = {

.type = response.type,

.error = ERROR\_EMPTY\_RESPONSE

};

response = empty;

}

GetSystemTime(&response.time);

result = send(client, (char\*)&response, sizeof(Response), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

closesocket(client);

return 1;

}

goto retry;

} break;

case REQ\_REACTIVE: {

for (;;) {

DWORDLONG value = 0;

response.error = 0 - get\_ram\_info(response.ts, &response.ts, NULL, &value, false);

value /= 1ULL << request.factor;

response.ram\_info.factor = request.factor;

response.ram\_info.value = value;

GetSystemTime(&response.time);

result = send(client, (char\*)&response, sizeof(Response), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

closesocket(client);

return 1;

}

}

} break;

default: {

error("unsupported request kind %d", request.kind);

response.error = ERROR\_UNSUPPORTED\_REQUEST\_KIND;

} break;

}

} break;

default: {

error("unknown request type %d", request.type);

response.error = ERROR\_UNSUPPORTED\_REQUEST\_TYPE;

} break;

}

GetSystemTime(&response.time);

result = send(client, (char\*)&response, sizeof(Response), 0);

if (result == SOCKET\_ERROR) {

error("unable to send (code %d)", WSAGetLastError());

}

}

result = shutdown(client, SD\_BOTH);

if (result == SOCKET\_ERROR) {

error("unable to shutdown client (code %d)", WSAGetLastError());

closesocket(client);

return 1;

}

return 0;

}

static DWORD WINAPI update\_ram\_info\_routine(LPVOID param) {

(void)param;

for (;;) {

update\_ram\_info();

Sleep(3000);

}

}

int main(int argc, char\* argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: coursework\_server\_b.exe <port>\n");

exit(1);

}

InitializeCriticalSection(&global\_ram\_critical\_section);

InitializeConditionVariable(&global\_ram\_condition\_variable);

WSADATA data;

int result = WSAStartup(MAKEWORD(2, 2), &data);

if (result) {

fatal\_error("unable to start WSA (code %d)", result);

}

struct addrinfo hints = { 0 };

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

hints.ai\_flags = AI\_PASSIVE;

struct addrinfo\* info;

result = getaddrinfo(NULL, argv[1], &hints, &info);

if (result) {

error("unable to get address info (code %d)", result);

WSACleanup();

return 1;

}

SOCKET sock = socket(info->ai\_family, info->ai\_socktype, info->ai\_protocol);

if (sock == INVALID\_SOCKET) {

error("unable to create socket (code %d)", WSAGetLastError());

freeaddrinfo(info);

WSACleanup();

return 1;

}

result = bind(sock, info->ai\_addr, (int)info->ai\_addrlen);

if (result == SOCKET\_ERROR) {

error("unable to bind socket (code %d)", WSAGetLastError());

freeaddrinfo(info);

closesocket(sock);

WSACleanup();

return 1;

}

freeaddrinfo(info);

result = listen(sock, SOMAXCONN);

if (result == SOCKET\_ERROR) {

error("unable to listen socket (code %d)", WSAGetLastError());

closesocket(sock);

WSACleanup();

return 1;

}

HANDLE thread = CreateThread(NULL, 0, update\_ram\_info\_routine, NULL, 0, NULL);

if (thread == INVALID\_HANDLE\_VALUE) {

error("unable to create routine thread (code %d)", WSAGetLastError());

closesocket(sock);

WSACleanup();

return 1;

}

for (;;) {

SOCKET client = accept(sock, NULL, NULL);

if (client == INVALID\_SOCKET) {

error("unable to accept client (code %d)", WSAGetLastError());

closesocket(client);

continue;

}

thread = CreateThread(NULL, 0, process\_client, (LPVOID)client, 0, NULL);

if (thread == INVALID\_HANDLE\_VALUE) {

error("unable to create thread for client (code %lu)", GetLastError());

}

}

}

Исходный код протокола:

#pragma once

#pragma warning(disable: 4201)

typedef struct {

char letter;

char fs[10];

} DriveInfo;

typedef enum {

REQ\_NONE,

REQ\_UNARY,

REQ\_TIMER,

REQ\_REACTIVE,

REQ\_COUNT

} RequestKind;

static char\* kind\_names[] = {

[REQ\_UNARY] = "unary",

[REQ\_TIMER] = "timer",

[REQ\_REACTIVE] = "reactive"

};

typedef enum {

TYPE\_NONE,

TYPE\_HARD\_DRIVE\_INFO,

TYPE\_CPU\_INFO,

TYPE\_FULL\_RAM,

TYPE\_AVAILABLE\_RAM,

TYPE\_COUNT

} RequestType, ResponseType;

static char\* type\_names[] = {

[TYPE\_HARD\_DRIVE\_INFO] = "hdinfo",

[TYPE\_CPU\_INFO] = "cpuinfo",

[TYPE\_FULL\_RAM] = "flram",

[TYPE\_AVAILABLE\_RAM] = "avram"

};

typedef struct {

RequestKind kind;

RequestType type;

uint32\_t ts;

union {

uint32\_t factor;

// ...

};

} Request;

static char\* factor\_names[] = {

[0] = "B",

[10] = "KB",

[20] = "MB",

[30] = "GB"

};

typedef enum {

ERROR\_NONE,

ERROR\_EMPTY\_RESPONSE,

ERROR\_UNSUPPORTED\_REQUEST\_KIND,

ERROR\_UNSUPPORTED\_REQUEST\_TYPE

} ResponseError;

typedef struct {

ResponseType type;

ResponseError error;

SYSTEMTIME time;

uint32\_t ts;

union {

struct {

uint8\_t count;

DriveInfo infos[26];

} hard\_drive\_info;

unsigned long logical\_cpu\_count;

struct {

uint32\_t factor;

uint64\_t value;

} ram\_info;

};

} Response;