Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра экономической информатики

КОМПЬЮТЕРНЫЕ СЕТИ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники для направления специальности 1-40 01 02-02 «Информационные системы и технологии (в экономике)» в качестве пособия

Минск БГУИР 2013

УДК 004.7(075.6)
ББК 32.973.202я73
К63

А в т о р ы:
В. Н. Комличенко, В. А. Федосенко, Т. М. Унучек,
Е. Н. Унучек, А. О. Комаровский

Р е ц е н з е н т ы:
кафедра управления информационными ресурсами
Академии управления при Президенте Республики Беларусь
(протокол №8 от 17.01.2013);

доцент кафедры систем автоматизированного проектирования
Белорусского национального технического университета,
кандидат технических наук, доцент И. Л. Ковалева

Компьютерные сети. Лабораторный практикум : пособие /
К63 В. Н. Комличенко [и др.]. – Минск : БГУИР, 2013. – 76 с.
ISBN 978-985-488-770-8.

Состоит из шести лабораторных работ, нацеленных на создание консольных приложений архитектуры «клиент-сервер» с конкретным типом сервера. Каждая лабораторная работа содержит основной теоретический материал по тематике работы, методические указания по созданию того или иного типа сервера, индивидуальные задания и контрольные вопросы.

УДК 004.7(075.6)
ББК 32.973.202я73

ISBN 978-985-488-770-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Дисциплина «Компьютерные сети» является базовой в цикле дисциплин, ориентированных на применение сетей и сетевых технологий в решении профессиональных задач, изучаемых студентами на последующих курсах обучения по специальности «Информационные системы и технологии (в экономике)».

Цель изучения данной дисциплины – овладение знаниями использования сетевых средств и базовых технологий программирования, а также приобретения основных навыков разработки программ сетевого взаимодействия, для использования при разработке сетевых информационных приложений.

Лабораторный практикум ориентирован на приобретение студентами знаний и базовых навыков разработки программ сетевого взаимодействия, получение навыков программной реализации протоколов среднего и верхнего уровня стека протоколов TCP/IP, а также опыта решения практических задач на основе технологии разделения функций между клиентом и сервером, что является основой организации архитектуры клиент-сервер.

В материалах практикума рассматриваются методика и принципы программной реализации последовательного, параллельного или параллельно-последовательного клиент-серверного взаимодействия. Значимость программных решений, представленных здесь, весьма существенная, поскольку программирование рассматривается с применением низкоуровневых программных абстракций, что позволяет достаточно хорошо изучить физическую основу процессов и методов взаимодействия узлов в компьютерных сетях. Рассмотренные схемы и алгоритмы сетевых взаимодействий дают представление о базовых концепциях, положенных в основу серверных разработок и организации синхронных взаимодействий приложений, формируют основу для понимания архитектурных принципов разработки современных распределенных информационных систем, способствуют более четкому и детальному пониманию основных алгоритмов и методов организации взаимодействия распределенных объектов и служб информационных систем.

ЛАБОРАТОРНАЯ РАБОТА №1

Создание последовательного сервера с установлением логического соединения TCP

Цель работы: изучить методы создания серверов с установлением логического соединения *TCP*, используя алгоритм последовательной обработки запросов.

1.1 Стек протоколов TCP/IP. История и перспективы стека TCP/IP

Протокол – набор правил и действий (очерёдности действий), позволяющий осуществлять соединение и обмен данными между двумя и более включёнными в сеть устройствами.

Стек протоколов – набор протоколов различных уровней, достаточный для организации взаимодействия в сети.

Transmission Control Protocol/Интернет Protocol (TCP/IP) – это промышленный стандарт стека протоколов, разработанный для глобальных сетей.

Агентство DARPA (Defense Advance Research Projects Agency) разработало стек TCP/IP (Transmission Control Protocol/Интернет Protocol) для объединения в сеть компьютеров различных подразделений министерства обороны США *(Department of Defence, DoD)* в 70-х годах прошлого века. В настоящее время стек TCP/IP является самым популярным средством организации составных сетей. До 1996 года бесспорным лидером был стек протоколов IPX/SPX компании Novell, но затем картина резко изменилась – стек TCP/IP по темпам роста числа установок намного стал опережать другие стеки, а с 1998 года вышел в лидеры и в абсолютном выражении.

Стандарты *TCP/IP* опубликованы в серии документов, названных *Request for Comments (RFC)*. Документы *RFC* описывают внутреннюю работу сети *Интернет*. Некоторые *RFC* описывают сетевые сервисы или протоколы и их реализацию, в то время как другие обобщают условия применения.

Лидирующая роль стека *TCP/IP* объясняется следующими его свойствами:

1. Это наиболее завершенный стандартный и в то же время популярный стек сетевых протоколов, имеющий многолетнюю историю.
2. Почти все большие сети передают основную часть своего трафика с помощью протокола *TCP/IP*.
3. Это метод получения доступа к сети *Интернет*.
4. Этот стек служит основой для создания *интранет* – корпоративной сети, использующей транспортные услуги сети *Интернет* и гипертекстовую технологию *WWW*.
5. Все современные операционные системы (ОС) поддерживают стек *TCP/IP*.

6. Это гибкая технология для соединения разнородных систем как на уровне транспортных подсистем, так и на уровне прикладных сервисов.
7. Это устойчивая масштабируемая межплатформенная среда для приложений клиент-сервер.

1.2 Структура стека TCP/IP. Краткая характеристика протоколов

Структура протоколов *TCP/IP* приведена на рисунке 1. Протоколы *TCP/IP* делятся на четыре уровня.

<table>
<tr><td>7</td><td rowspan="2">HTTP, HTTPs</td><td rowspan="2">SNMP</td><td rowspan="2">FTP</td><td rowspan="2">telnet</td><td rowspan="2">SMTP</td><td rowspan="2">TFTP</td><td rowspan="2">I</td></tr>
<tr><td>6</td></tr>
<tr><td>5</td><td colspan="5" rowspan="2">TCP</td><td rowspan="2">UDP</td><td rowspan="2">II</td></tr>
<tr><td>4</td></tr>
<tr><td>3</td><td>IP</td><td colspan="2">ICMP</td><td>RIP</td><td>OSPF</td><td>ARP
RARP</td><td>III</td></tr>
<tr><td>2</td><td colspan="6" rowspan="2">Не регламентируется
Ethernet, Token Ring, FDDI, X.25, SLIP, PPP</td><td rowspan="2">IV</td></tr>
<tr><td>1</td></tr>
<tr><td colspan="7">Уровни модели OSI</td><td>Уровни стека TCP/IP</td></tr>
</table>

Рисунок 1 – Стек *TCP/IP*

Самый нижний (уровень IV) соответствует физическому и канальному уровням модели *OSI*. Этот уровень в протоколах *TCP/IP* не регламентируется, но поддерживает все популярные стандарты физического и канального уровня: для локальных сетей это *Ethernet, Fast Ethernet, Gigabit Eyhernet, 10GEthernet, 100GEthernet, 100VG-AnyLAN, Token Ring*, *FDDI*, для глобальных сетей – протоколы соединений «точка-точка» *SLIP* и *PPP*, протоколы территориальных сетей с коммутацией пакетов *X.25, Frame Relay*. Разработана также специальная спецификация, определяющая использование технологии *ATM* в качестве транспорта канального уровня. Обычно при появлении новой технологии локальных или глобальных сетей она быстро включается в стек *TCP/IP* за счет разработки соответствующего *RFC*, определяющего метод инкапсуляции пакетов *IP* в ее кадры.

Следующий уровень (уровень III) – это уровень межсетевого взаимодействия, который занимается передачей пакетов с использованием различных транспортных технологий локальных и территориальных сетей, линий специальной связи и т. п.

В качестве основного протокола сетевого уровня (в терминах модели *OSI*) в стеке используется протокол *IP*, который изначально проектировался как про-

токол передачи пакетов в составных сетях, состоящих из большого количества локальных сетей, объединенных как локальными, так и глобальными связями. Поэтому протокол *IP* хорошо работает в сетях со сложной топологией, рационально используя наличие в них подсистем и экономно расходуя пропускную способность низкоскоростных линий связи. Протокол *IP* является дейтаграммным протоколом, то есть он не гарантирует доставку пакетов до узла назначения, но старается это сделать.

К уровню межсетевого взаимодействия относятся и все протоколы, связанные с составлением и модификацией таблиц маршрутизации, такие как протоколы сбора маршрутной информации *RIP* (Routing Интернет Protocol) и *OSPF (Open Shortest Path First)*, а также протокол межсетевых управляющих сообщений *ICMP (Internet Control Message Protocol)*. Последний протокол предназначен для обмена информацией об ошибках между маршрутизаторами сети и узлом – источником пакета. С помощью специальных пакетов *ICMP* сообщается о невозможности доставки пакета, о превышении времени жизни или продолжительности сборки пакета из фрагментов, об аномальных величинах параметров, об изменении маршрута пересылки и типа обслуживания, о состоянии системы и т.п.

Следующий уровень (уровень II) называется основным. На этом уровне функционируют протокол управления передачей *TCP* и протокол дейтаграмм пользователя *UDP*. Протокол *TCP* обеспечивает надежную передачу сообщений между удаленными прикладными процессами за счет образования виртуальных соединений. Протокол *UDP* обеспечивает передачу прикладных пакетов дейтаграммным способом, как и *IP*, но выполняет только функции связующего звена между сетевым протоколом и многочисленными прикладными процессами.

Верхний уровень (уровень I) называется прикладным. За долгие годы использования в сетях различных стран и организаций стек *TCP/IP* накопил большое количество протоколов и сервисов прикладного уровня. К ним относятся такие широко используемые протоколы, как протокол копирования файлов *FTP*, протокол эмуляции терминала *telnet*, почтовый протокол *SMTP*, используемый в электронной почте сети *Интернет*, гипертекстовые сервисы доступа к удаленной информации, такие как *WWW,* и многие другие. Остановимся несколько подробнее на некоторых из них.

Протокол пересылки файлов *FTP (File Transfer Protocol)* реализует удаленный доступ к файлу. Для того чтобы обеспечить надежную передачу, *FTP* использует в качестве транспорта протокол с установлением соединений – *TCP*. Кроме пересылки файлов протокол *FTP* предлагает и другие услуги.

В стеке *TCP/IP* протокол *FTP* предлагает наиболее широкий набор услуг для работы с файлами, однако он является и самым сложным для программирования. Приложения, которым не требуются все возможности *FTP*, могут использовать другой, более экономичный протокол – простейший протокол пересылки файлов *TFTP (Trivial File Transfer Protocol)*. Этот протокол реализует

только передачу файлов, причем в качестве транспорта используется более простой, чем *TCP*, протокол без установления соединения – *UDP*.

Протокол *telnet* обеспечивает передачу потока байтов между процессами, а также между процессом и терминалом. Наиболее часто этот протокол используется для эмуляции терминала удаленного компьютера. При использовании сервиса *telnet* пользователь фактически управляет удаленным компьютером так же, как и локальный пользователь, поэтому такой вид доступа требует хорошей защиты.

Протокол *SNMP (Simple Network Management Protocol)* используется для организации сетевого управления. Изначально протокол *SNMP* был разработан для удаленного контроля и управления маршрутизаторами *Интернет*, которые традиционно часто называют также шлюзами. С ростом популярности протокол *SNMP* стали применять и для управления любым коммуникационным оборудованием – концентраторами, мостами, сетевыми адаптерами и т.д.

1.3 Протокол TCP

Протокол *TCP (Transmission Control Protocol)* работает, как и протокол *UDP*, на транспортном уровне. Он обеспечивает надежную транспортировку данных между прикладными процессами путем установления логического соединения.

Единицей данных протокола *TCP* является сегмент. Информация, поступающая к протоколу *TCP* в рамках логического соединения от протоколов более высокого уровня, рассматривается протоколом *TCP* как неструктурированный поток байтов. Поступающие данные буферизуются средствами *TCP*. Для передачи на сетевой уровень из буфера «вырезается» некоторая непрерывная часть данных, называемая сегментом.

Не все сегменты, посланные через соединение, будут одного и того же размера, однако оба участника соединения должны договориться о максимальном размере сегмента, который они будут использовать. Этот размер выбирается таким образом, чтобы при упаковке сегмента в *IP*-пакет он помещался туда целиком, то есть максимальный размер сегмента не должен превосходить максимального размера поля данных *IP*-пакета. В противном случае пришлось бы выполнять фрагментацию, то есть делить сегмент на несколько частей, для того чтобы он вместился в *IP*-пакет.

Аналогичные проблемы решаются и на сетевом уровне. Для того чтобы избежать фрагментации, должен быть выбран соответствующий максимальный размер *IP*-пакета. Однако при этом должны быть приняты во внимание максимальные размеры поля данных кадров (*MTU*) всех протоколов канального уровня, используемых в сети. Максимальный размер сегмента не должен превышать минимальное значение на множестве всех *MTU* составной сети.

1.4 Установление TCP-соединений

В протоколе *TCP*, как и в *UDP*, для связи с прикладными процессами используются порты. Номера портам присваиваются следующим образом: име-

ются стандартные, зарезервированные номера (например, номер 21 закреплен за сервисом *FTP*, 23 – за *telnet*), а менее известные приложения пользуются произвольно выбранными локальными номерами.

Как говорилось выше, для организации надежной передачи данных предусматривается установление *логического соединения* между двумя прикладными процессами. В рамках соединения осуществляется обязательное подтверждение правильности приема для всех переданных сообщений, и при необходимости выполняется повторная передача. Соединение в *TCP* позволяет вести передачу данных одновременно в обе стороны, то есть полнодуплексную передачу.

Соединение в протоколе *TCP* идентифицируется парой полных адресов обоих взаимодействующих процессов (оконечных точек). Адрес каждой из оконечных точек включает *IP*-адрес (номер сети и номер компьютера) и номер порта. Одна оконечная точка может участвовать в нескольких соединениях.

Установление соединения выполняется в следующей последовательности:

1. При установлении соединения одна из сторон является инициатором. Она посылает запрос к протоколу *TCP* на открытие порта для передачи (*active open*).
2. После открытия порта протокол *TCP* на стороне процесса-инициатора посылает запрос процессу, с которым требуется установить соединение.
3. Протокол *TCP* на приемной стороне открывает порт для приема данных (*passive open*) и возвращает квитанцию, подтверждающую прием запроса.
4. Для того чтобы передача могла вестись в обе стороны, протокол на приемной стороне также открывает порт для передачи (*active port*) и также передает запрос к противоположной стороне.
5. Сторона-инициатор открывает порт для приема и возвращает квитанцию. Соединение считается установленным. Далее происходит обмен данными в рамках данного соединения.

Для *UDP* соединений существует своя схема взаимодействия между приложениями, которая будет подробно рассмотрена в следующих лабораторных работах.

Схема работы сервера определяется не только протоколом взаимодействия с клиентом (*TCP* или *UDP*), но и тем, какой механизм реализован на сервере по обработке клиентских запросов (последовательный или параллельный). В данном практикуме будут рассмотрены следующие типы серверов:

- последовательный сервер с установлением логического соединения;
- последовательный сервер без установления логического соединения;
- параллельный многопоточный сервер с установлением логического соединения;
- параллельный многопроцессный сервер с установлением логического соединения;

- однопотоковый параллельный сервер с установлением логического соединения;
- параллельный сервер с установлением логического соединения, использующий пул потоков.

Определить, какой из типов серверов использовать, достаточно непросто, так как для оценки работы сервера следует анализировать несколько параметров, как минимум это, время реакции, максимальное число обслуженных запросов в секунду (интенсивность), число отказов обслуживания, и на какой интенсивности они начинают возникать. Всё определяется теми задачами, которые решает сервер, и тем окружением, в котором он функционирует. Это могут быть и области, где наилучшим решением окажется простейший последовательный сервер и такие, где это решение неприемлемо. Кроме того, следует учитывать и такие аспекты, как трудоёмкость реализации, потребление ресурсов (в частности оперативной памяти), простота отладки, сопровождения и др.

1.5. Алгоритм работы последовательного сервера с установлением логического соединения

Покажем обобщенный алгоритм работы последовательного сервера с установлением логического соединения:

1. Создать сокет и установить связь с локальным адресом.
2. Перевести сокет в пассивный режим, подготавливая его для использования сервером.
3. Принять из сокета следующий запрос на установление соединения и получить новый сокет для соединения.
4. Считывать в цикле запросы от клиента, формировать ответы и отправлять клиенту.
5. После завершения обмена данными с конкретным клиентом закрыть соединение и возвратиться к этапу 3 для приема нового запроса на установление соединения.

На рисунке 2 показана схема организации работы последовательного сервера с установлением логического соединения.

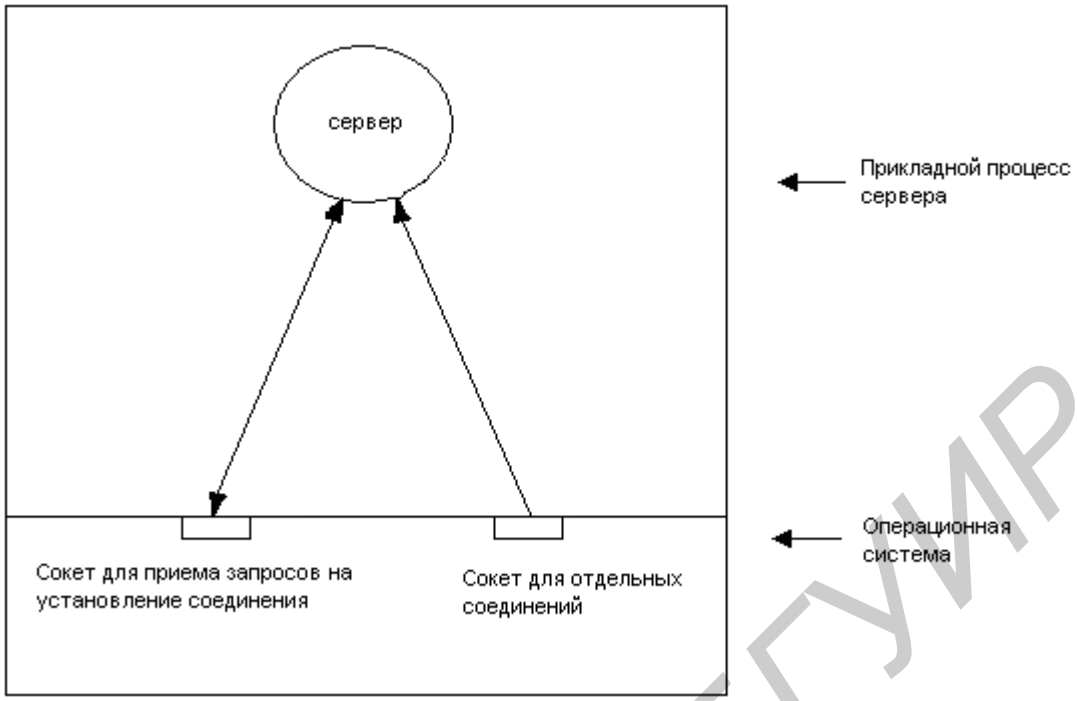

Рисунок 2 – Схема организации работы последовательного сервера с установления логического соединения

1.6 Методические указания по созданию последовательного сервера с установлением логического соединения (*TCP*)

В качестве примера приведем следующую задачу.

Осуществить взаимодействие клиента и сервера на основе протокола TCP. Функционирование клиента и сервера реализовать следующим образом: клиент посылает произвольный набор символов серверу и получает назад количество символов «а» в этом наборе.

Необходимо написать два проекта на *C* – клиент и сервер. Начнем с серверной части.

Серверная часть

При разработке приложений для клиента и сервера для обмена структурами данных или пакетами используются сокеты. Сокет – это абстрактный объект для обозначения одного из концов сетевого соединения. Он предназначен для создания механизма обмена данными. Реализация сокетов осуществляется в *API WinSock*.

В версии 1.1 WinSock любого поставщика имеется библиотека *WSOCK32.DLL* (или *winsock .dll* для 16-разрядных операционных систем), позволяющая реализовать программный интерфейс *WinSock*. Интерфейс версии 2 в системе Windows поддерживается одной динамической библиотекой *WS2_32. dll*, которая для обслуживания различных сетей может использовать протоколы и системы распознавания имен различных разработчиков. Библиотека *WS2_32.dll* поддерживает как функции *WinSock 1.1*, так и ряд дополнительных функций, впервые появившихся в спецификации *WinSock 2*. Данную библиоте-

ку необходимо подключить к проекту, выполненному в *VC++*: *Project –>Settings – вкладка Links* – к списку подключаемых библиотек через пробел добавляем *ws2_32.lib* .

В тексте программы этот интерфейс разработки приложений подключается с помощью директивы *#include*:

```
#include <winsock2.h>
```

Кроме того, подключим уже известные заголовочные файлы :

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
```

Для того чтобы можно было использовать интерфейс программирования *WinSock*, его необходимо инициализировать с помощью функции *WSAStartup(wVersionRequested,&wsaData)*.

Первый параметр функции *WSAStartup()* – это значение типа *word*, которое определяет максимальный номер версии *WinSock*, доступный приложению. Первая цифра версии находится в младшем байте, вторая – в старшем.

Функция *WSAStartup()* возвращает значение *wsasysnotready*, если динамическая библиотека поддержки *WinSock* или соответствующая подсистема сети не инициализирована, инициализирована некорректно или не найдена. Кроме того, с помощью этой функции приложение сообщает системе версию *WinSock*, которая должна использоваться. Как правило, при вызове функции *WSAStartup()* необходимо указывать максимальный допустимый номер версии. Если он меньше, чем версии, поддерживаемые данной динамической библиотекой, функция *WSAStartup()* возвратит значение *wsavernotsupported*.

Второй параметр – структура *wsaData* – содержит номер версии, которая должна использоваться (поле *wVersion*), максимальный номер версии, поддерживаемый данной библиотекой (поле *wHighVersion*), текстовые строки с описанием реализации *WinSock*, максимальное число сокетов, доступных процессу и максимально допустимый размер дейтаграмм.

Инициализация *WinSock* с помощью функции *WSAStartup()* в нашей программе описывается так:

```
int main(){
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested=MAKEWORD(2,2);
    WSAStartup(wVersionRequested,&wsaData);
```

В данной работе нас интересуют сокеты потоков (`SOCK_STREAM`), которые позволяют гарантировать бесперебойную доставку данных в нужном порядке и без дублирования. В *TCP/IP*-реализации *WinSock* сокеты потоков ис-

пользуют протокол *TCP* (*Transmission Protocol*). Сокеты потоков обеспечивают пересылку больших объемов данных без потерь и нарушения порядка. Кроме того, при закрытии соединения приложения получат извещение об этом событии.

Для создания сокета используется функция *socket(domain,type,protocol)*. Она принимает три параметра: домен, тип сокета и протокол. Домен – это абстракция, подразумевающая конкретную структуру адресации и протоколы, определяющие типы сокетов внутри домена. Примерами коммуникационных доменов могут быть: *UNIX* домен, *Интернет* домен, и т.д. В *Интернет* домене сокет – это комбинация *IP*-адреса и номера порта, которая однозначно определяет отдельный сетевой процесс во всей глобальной сети *Интернет*. Два сокета, один для хоста-получателя, другой – для хоста-отправителя, определяют соединение для протоколов, ориентированных на установление связи, таких, как *TCP*.

Вызов функции *socket()* выглядит следующим образом:

```
SOCKET  s = socket(AF_INET, SOCK_STREAM, 0);
```

Первый параметр означает, что с этим сокетом будут использоваться адреса *Интернет*; следующие два аргумента задают тип создаваемого сокета и протокол обмена данными через него. В приведенном примере создается сокет потока, использующий протокол *TCP*.

Если третий параметр функции *socket()* сделать равным нулю, протокол будет выбран автоматически в зависимости от семейства адресов и типа сокета. Можно явно указать константы:

IPPROTO_UDP – протокол *UDP* (смотри лабораторную работу №2),

IPPROTO_TCP – протокол *TCP/IP*.

Если функция *socket()* выполняется успешно, она возвращает дескриптор нового сокета. Если же ее работа завершается аварийно, возвращается значение 0, и для получения подробной информации об ошибке необходимо вызвать функцию *WSAGetLastError ( )*.

Для связывания конкретного адреса с сокетом используется функция *bind (s, addr, addrlen)*. В нее передается дескриптор сокета, указатель на структуру адреса и длина этой структуры. Дескриптор сокета – это значение, которое возвращает функция *socket()*. Структура адреса – это структура типа *sockaddr_in*.

```
struct sockaddr_in local;
local.sin_family=AF_INET;
local.sin_port=htons(1280);
local.sin_addr.s_addr=htonl(INADDR_ANY);
int c=bind(s,(struct sockaddr*)&local,sizeof(local));
```

В поле *sin_addr* структуры *sockaddr_in* хранится физический *IP*-адрес компьютера в формате структуры *in_addr*, описанной в заголовочном файле

winsock2.h. Вместо поля *s_addr* можно подставлять *INADDR_ANY*. это позволяет сокету посылать или принимать данные через любой *IP*-адрес данного компьютера. Обычно компьютер имеет только один *IP*-адрес, хотя в принципе на нем может быть установлено несколько сетевых адаптеров, каждый со своим *IP*-адресом. Если сокет должен использовать только один из них, его необходимо указать явно. Для этого нередко используется функция *inet_addr("…")* , которая принимает в качеств аргумента *ASCII*-строку десятичной нотации *IP*-адреса с точкой и возвращает переменную типа *u_long*, содержащую этот адрес в формате поля *s_addr*. Кроме нее, существует функция *inet_ntoa(address),* которая выполняет обратное преобразование, принимая переменную типа *u_long* и возвращая адрес в виде *ASCII*-строки.

Поле *sin_family* всегда имеет значение *AF_INET*. Поле *sin_port* определяет порт, который будет ассоциирован с сокетом.

Для привязки приложение может использовать любой номер порта от 1 до 65535, хотя этот диапазон обычно делится на следующие поддиапазоны:

0 – не используется. Если передать 0 в качестве номера порта, будет автоматически выбран используемый порт с номером между 1 024 и 5 000.

1 – 255 – зарезервированы для сетевых служб: *FTP*, *telnet*, *finger* и т.д.

256 – 1 023 – зарезервированы для других служб общего назначения, например функций маршрутизации.

1 024 – 4 999 – служат для портов клиентов. Обычно сокеты приложений-клиентов используют номера портов именно из этого диапазона.

5 000 – 65 535. Используются для определяемых пользователем портов приложений-серверов.

Вместо простого присвоения констант полей *sin_port* и *sin_addr* использовалось преобразование типов с помощью функций *htons(n)* и *htonl(n)* Эти функции предназначены для изменения порядка следования байтов в параметрах порта и адреса, для преобразования их в общий сетевой формат для 16-разрядных и 32-разрядных значений соответственно.

После создания сокета и привязки его к адресу необходимо каким-то образом установить соединение с клиентом. Для этого используется функция *listen (s, l)* , которая помещает сокет в состояние прослушивания:

```
int r=listen(s,5);
```

Вызов этой функции инициирует ожидание запроса клиента на открытие соединения. Параметр *l* содержит количество запросов, которое должно поступить для того, чтобы приложение согласилось установить соединение. Например, если этот параметр равен 2 и приложение по каким-то причинам отказалось открыть соединение, третий клиент, который попытается подключиться к серверу, получит код ошибки *wsaeconnrefused*. Первые два запроса будут отправлены в очередь для их последующей обработки сервером.

При получении запроса клиента открытие соединения выполняется с помощью функции *accept()* :

```
SOCKET accept (SOCKET s, struct sockaddr FAR * addr, int FAR*
addrlen)
```

Как обычно, в качестве первого параметра передается сокет, ожидающий запроса. Второй и третий параметры используются для получения адреса сокета клиента, который запрашивает соединение. Если соединение открывается успешно, функция *accept()* возвращает дескриптор на новый сокет, который будет использоваться для управления новым соединением. Если произошла ошибка, функция *accept()* возвращает код *invalid_socket*, и для получения более подробной информации об ошибке необходимо вызвать функцию *WSAGetLastError()* .

Исходный сокет продолжит ожидание запросов на новые соединения, которые затем открываются снова с помощью функции *accept()*. Каждое открытое соединение управляется отдельным сокетом, дескриптор которого возвращается из этой функции.

В нашем примере это выглядит так:

```
while (true){
     char buf[255],res[100],b[255],*Res;
     //структура определяет удаленный адрес,
     //с которым  соединяется сокет
     sockaddr_in remote_addr;
     int size=sizeof(remote_addr);
     SOCKET s2=accept(s,(struct sockaddr*)&remote_addr,&size);
```

Для выполнения задачи нам необходимо осуществлять прием и передачу данных. Ввод исходной строки выполнит клиент и передаст ее серверу, чтобы тот проанализировал ее и отослал назад клиенту количество букв «а» в этой строке.

Для приема данных через сокет потока используется функция *recv*(). Вот ее прототип: *int recv (SOCKET s, char FAR* buf, int len, int flags);* Параметры *buf* и *len* определяют соответственно буфер для приема данных и его длину. Параметр *flags* может принимать значения *MSG_OOB* для приема привилегированных данных или *MSG_ PEEK* для заполнения буфера без удаления данных из входной очереди, но, как правило, мы пишем его равным нулю.

Если во входной очереди находятся данные для сокета, функция *recv()* возвращает количество прочитанных байтов, которое равно объему доступных данных во входной очереди и не превосходит значения *len*. При корректном закрытии соединения возвращается значение 0, а при аварийном – значение SOCKET_ERROR. Для определения точного кода ошибки необходимо вызвать функцию *WSAGetLastError()* .

Пересылка данных выполняется с помощью функции *send*() :

```
int send (SOCKET s, const char FAR *buf, int len, int flags)
```

Функция *send()* принимает в качестве аргументов указатель на буфер, содержащий пересылаемые данные, и его длину, а также параметр *flags*. Если этот параметр равен *msg_dontroute*, в пересылаемый набор данных не включается информация о маршрутизации; если его значение равно *msg_oob*, посылается поток привилегированных (*out-of-band*) данных.

Объем данных, пересылаемых одним вызовом функции *send()* , не должен превышать размера пакета, максимально допустимого в данной сети. При попытке пересылки большего объема данных функция *send()* завершится аварийно, а функция *WSAGetLastError()* возвратит код ошибки *WSAEMSGSIZE*.

Работу с одним клиентом поместим в цикл, чтобы была возможность вводить несколько строк.

```
while (recv(s2,b,sizeof(b),0)!=0){

     int i=0;
     for (unsigned j=0;j<=strlen(b);j++)
          if (b[j]=='a') i++;

     _itoa(i,res,10);

     Res=new char[strlen(res)+1];
     strcpy(Res,res);
     Res[strlen(res)]='\0';


     //Посылает данные на соединенный сокет
     send(s2,Res,sizeof(Res)-2,0);

}
```

Для завершения работы сокета его необходимо закрыть с помощью функции *closesocket*() : *int closesocket (SOCKET s)*. Эта функция принимает единственный аргумент – дескриптор закрываемого сокета, но ее поведение определяется также параметрами сокета, установленными с помощью функции *setsockopt()*. Текущие параметры можно узнать, вызвав функцию *getsockopt()*. Результат работы функции *closesocket()* определяется параметрами *SO_LINGER* и *SO_DONTLINGER*.

Если параметр *SO_DONTLINGER* равен *TRUE*, функция *closesocket()* возвратит значение «немедленно», но перед закрытием сокета будет предпринята попытка пересылки всех оставшихся данных. Обычно это называется корректным закрытием.

Если параметр *SO_LINGER* равен *TRUE* и установлена ненулевая задержка, также выполняется корректное закрытие с попыткой пересылки всех оставшихся в буфере данных, но функция *closesocket()* не возвращает значения до тех пор, пока не будут пересланы все данные или пока не истечет срок задерж-

ки. Если параметр *SO_LINGER* равен *TRUE* и задержка равна нулю, сокет закрывается немедленно, а все оставшиеся в буфере данные теряются.

Закроем сокет в нашей программе:

```
            closesocket(s2);
}
```

Другие способы закрытия сокета. Если сокет больше не используется, процесс может закрыть его с помощью функции *close (s)*, вызвав ее с соответствующим дескриптором сокета: *close(s)*.

Если данные были ассоциированы с сокетом, обещающим доставку (сокет типа *stream*), система будет пытаться осуществить передачу этих данных. Тем не менее, по истечении довольно-таки длительного промежутка времени, если данные все еще не доставлены, они будут отброшены. Если пользовательский процесс желает прекратить любую передачу данных, он может сделать это с помощью вызова *shutdown* на данном сокете для его закрытия. Вызов *shutdown* вызывает «моментальное» отбрасывание всех стоящих в очереди данных. Формат вызова следующий: *shutdown(s, how)*, где *how* имеет одно из следующих значений:

0 – если пользователь больше не желает читать данные;

1 – если данные больше не будут посылаться;

2 – если данные не будут ни посылаться, ни получаться.

Завершая программу, нужно прекратить работу с *WinSock DLL*, вызвав функцию:

```
        WSACleanup();
}
```

Клиентская часть

Ниже приведена программа клиента.

```
#include <winsock2.h>
#include <iostream.h>
#include <stdlib.h>

int main(){

        WORD wVersionRequested;
        WSADATA wsaData;
        wVersionRequested=MAKEWORD(2,2);
        WSAStartup(wVersionRequested,&wsaData);

        struct sockaddr_in peer;
        peer.sin_family=AF_INET;
        peer.sin_port=htons(1280);
        / /т.к. клиент и сервер на одном компьютере,
        / / пишем адрес 127.0.0.1
```

```
        peer.sin_addr.s_addr=inet_addr("127.0.0.1");

        SOCKET s=socket(AF_INET,SOCK_STREAM,0);

        connect(s,(struct sockaddr*) &peer,sizeof(peer));

        char buf[255],b[255];
        cout<<"Enter the string, please"<<endl;
        cin.getline(buf,100,'\n');

        send(s,buf,sizeof(buf),0);
        if (recv(s,b,sizeof(b),0)!=0){
             b[strlen(b)]='\0'; / /Удаление ненужных символов
                               / / в конце строки
             cout<<b<<endl;
        }

        closesocket(s);

        WSACleanup();

        return 0;
}
```

Клиентская часть использует функции, которые описаны ранее. Новая функция, которая не вызывается в серверной части, – *connect (s, addr, addrlen).*С помощью этой функции приложение-клиент посылает запрос на открытие соединения. Параметры *addr*, *addrlen* используются для указания адреса и порта, к которому необходимо подсоединиться. Структура *sockaddr*, передаваемая в функцию *connect()*, должна быть идентичной структуре, передаваемой в функцию *bind()* на сервере.

1.7 Индивидуальные задания

Разработать приложение, реализующее архитектуру «клиент-сервер». Необходимо реализовать последовательный сервер с установлением логического соединения (*TCP*). Логику взаимодействия клиента и сервера реализовать следующим образом:

1. Клиент посылает два числа серверу и одну из математических операций: «*», «/», «+», «–» ,– сервер соответственно умножает, делит, складывает либо вычитает эти два числа и посылает ответ назад клиенту.
2. Клиент посылает слово серверу, сервер возвращает назад в обратном порядке следования букв это слово клиенту.
3. Клиент посылает два числа серверу *m* и *n*, сервер возвращает *m!+n!* этих чисел назад клиенту.
4. Клиент посылает два слова серверу, сервер их сравнивает и возвращает «истина», если они одинаковы по количеству и порядку следования в них букв, и «ложь»– при невыполнении хотя бы одного из этих условий.

5. Клиент посылает произвольный набор латинских букв серверу и получает их назад упорядоченными по алфавиту.

6. Клиент посылает серверу произвольный набор символов, сервер замещает каждый четвертый символ на «%».

7. Сервер генерирует прогноз погоды на неделю. Клиент посылает день недели и получает соответствующий прогноз.

8. Клиент посылает серверу произвольные числа и получает назад количество чисел, кратных трем.

9. Клиент посылает серверу символьную строку, содержащую пробелы, и получает назад ту же строку, но в ней между словами должен находиться только один пробел.

10. Клиент посылает серверу слово. Сервер определяет, является ли это слово палиндромом (*палиндром* – слово, читающееся одинаково как слева направо и справа налево).

11. Клиент посылает серверу два числа и получает назад НОД (наибольший общий делитель) этих чисел.

12. Клиент посылает серверу число от 0 до 10 и получает назад название этого числа прописью.

13. Клиент посылает серверу координаты точки X и У в декартовой системе координат. Сервер определяет, в какой координатной четверти находится данная точка и посылает результат назад клиенту.

14. Клиент посылает серверу координаты прямоугольной области и точки в декартовой системе координат. Сервер определяет, лежит ли данная точка в прямоугольной области, и посылает результат назад клиенту.

15. Клиент посылает серверу шестизначный номер билета. Сервер определяет, является ли этот билет «счастливым». «Счастливым» называется такой билет, у которого сумма первых трех цифр равна сумме последних трех. Сервер посылает результат назад клиенту.

1.8 Контрольные вопросы

1. Какая технология называется межсетевым обменом (*Интернетworking*)?

2. Объясните понятие «протоколы» в контексте технологий обмена данными. Что они включают? Приведите примеры.

3. Назовите отличия *TCP/IP* от других средств передачи данных.

4. Дайте определение понятию «сокет».

5. Опишите функцию, которая используется для приема данных через сокет потока (протокол *TCP*).

6. Назовите функцию, используемую для создания сокета. Опишите ее параметры.

7. Опишите функцию, которая используется для пересылки данных через сокет потока (протокол *TCP*).

8. Что возвращает функция *accept()* в том случае, если соединение открывается успешно?

9. Назовите функцию, которая используется в приложении-клиенте для посылки запроса на открытие соединения. Опишите ее параметры.

ЛАБОРАТОРНАЯ РАБОТА №2

Создание последовательного сервера без установления логического соединения UDP

Цель работы: изучить методы создания серверов без установления логического соединения *UDP*, используя алгоритм последовательной обработки запросов.

2.1 Протокол UDP

Задачей протокола транспортного уровня *UDP (User Datagram Protocol)* является передача данных между прикладными процессами без гарантий доставки, поэтому его пакеты могут быть потеряны, продублированы или прийти не в том порядке, в котором они были отправлены.

Каждый коммуникационный протокол оперирует с некоторой единицей передаваемых данных. Единицу данных протокола *UDP* называют *дейтаграммой (datagram)*. Протокол *UDP* является простейшим дейтаграммным протоколом, который используется в том случае, когда задача надежного обмена данными либо вообще не ставится, либо решается средствами более высокого уровня – системными прикладными службами или пользовательскими приложениями.

В стеке протоколов *TCP/IP* протокол *UDP* обеспечивает основной механизм, используемый прикладными программами для передачи дейтаграмм другим приложениям. *UDP* предоставляет протокольные порты, используемые для различения нескольких процессов, выполняющихся на одном компьютере. Помимо посылаемых данных каждое *UDP*-сообщение содержит номер порта-приемника и номер порта-отправителя, делая возможным для программ *UDP* на машине-получателе доставку сообщения соответствующему реципиенту, а для получателя – посылку ответа соответствующему отправителю. Этот протокол обеспечивает ненадежную доставку данных «по возможности» в отличие от протокола *TCP*, обеспечивающего гарантированную доставку. *UDP* не использует подтверждения прихода сообщений, не упорядочивает приходящие сообщения и не обеспечивает обратной связи для управления скоростью передачи информации между машинами. Поэтому *UDP*-сообщения могут быть потеряны, размножены или приходить не по порядку. Кроме того, пакеты могут приходить раньше, чем получатель сможет обработать их.

2.2 Сокеты дейтаграмм

Сокеты дейтаграмм (datagram sockets) – это средства поддержки не очень надежного обмена пакетами. Ненадежность в данном контексте означает отсутствие гарантии их доставки по назначению в требуемом порядке. Фактически один и тот же пакет дейтаграммы может быть доставлен несколько раз.

Хотя *WinSock* поддерживает и другие протоколы, во многих случаях, на-

пример, при обмене данными между двумя процессами на одном и том же компьютере или между двумя компьютерами в локальной сети с небольшой нагрузкой, исключена путаница, неправильное название пакетов или доставка их не по адресу. Однако полной гарантии от подобных неприятностей приложение не обеспечивает.

Если же приложение управляет обменом данными по более сложной и загруженной сети, ненадежный характер сокетов дейтаграмм проявится очень быстро. При отсутствии в приложении обработки ошибок оно просто не справится с задачей. Тем не менее сокеты дейтаграмм очень полезны для пересылки данных, состоящих из отдельных пакетов или записей. Они также являются удобным средством рассылки циркулярных пакетов по нескольким адресам одновременно.

Примерами сетевых приложений, использующих *UDP*, являются *NFS* (*Network File System* – сетевая файловая система) и *SNMP* (*Simple Network Management Protocol* – простой протокол управления сетью).

2.3 Алгоритм работы последовательного сервера без установления логического соединения

Опишем обобщенный алгоритм работы последовательного сервера без установления логического соединения:

1. Создать сокет сервера (связав его с доступным портом и локальным адресом).

2. Считывать в цикле запросы от клиента, формировать ответы и отправлять клиенту в соответствии с прикладным протоколом.

На рисунке 3 показана схема организации работы последовательного сервера без установления логического соединения. Требуется только один поток выполнения, который обеспечивает взаимодействие сервера со многими клиентами с использованием одного сокета.

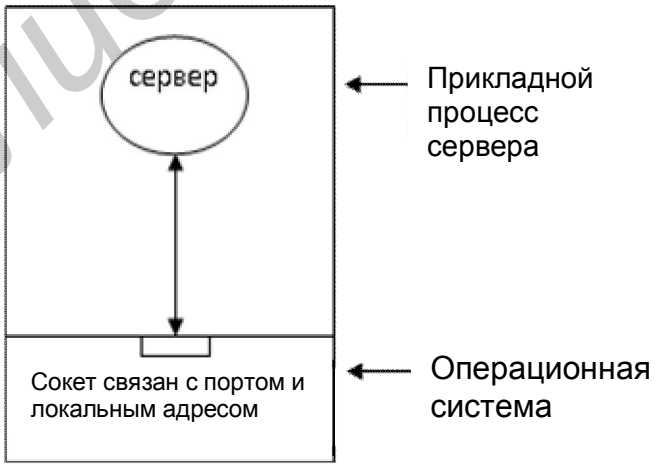

Рисунок 3 – Схема организации работы последовательного сервера без установления логического соединения

2.4 Методические указания по созданию последовательного сервера без установления логического соединения UDP

Для того чтобы продемонстрировать особенности работы протокола *UDP,* рассмотрим следующую задачу.

Осуществить взаимодействие клиента и сервера на основе протокола *UDP*. Функциональные возможности клиента реализовать следующим образом: клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Функциональные возможности сервера реализовать следующим образом: сервер, получив эту строку, должен поменять в ней местами символы на четных и нечетных позициях. Результат возвратить назад клиенту.

Так же как и в предыдущей лабораторной работе, создадим два проекта – клиент и сервер.

Серверная часть

```
#include<winsock2.h>
#include<iostream.h>
#include<stdlib.h>
#include<process.h>

void main(void){
     WORD wVersionRequested;
     WSADATA wsaData;
     int err;
     wVersionRequested=MAKEWORD(2,2);
     err = WSAStartup(wVersionRequested,&wsaData);
```

Вызов функции *socket()* выглядит следующим образом:

```
     SOCKET s;
     s = socket(AF_INET,SOCK_DGRAM,0);
```

Так как в данном случае создается сокет дейтаграмм, использующий протокол *UDP*, вторым аргументом, задающим тип создаваемого сокета, должен быть *SOCK_DGRAM.*

```
     struct sockaddr_in ad;
     ad.sin_port = htons(1024);
     ad.sin_family = AF_INET;
     ad.sin_addr.s_addr = 0;//подставляет подходящий ip
     bind(s,(struct sockaddr*) &ad,sizeof(ad));

     char b[200],tmp='\0';
          int l;
     l = sizeof(ad);
```

После создания сокета в приложении-сервере и привязки его к определенному адресу и порту можно принимать данные от приложения-клиента. Это делается с помощью функции *recvfrom()*, имеющей следующий прототип:

```
int recvfrom(SOCKET s,char FAR * buf,int len,int flags,
struct sockaddr FAR *from,int FAR *fromlen);
```

Первый параметр – это дескриптор сокета, возвращаемый функцией *socket()*; за ним идут указатель на буфер для приема новой дейтаграммы и длина буфера. Параметр *flags* может быть равен *msg_peek* для заполнения буфера таким образом, что дейтаграмма остается во входной очереди. Последние два параметра используются для получения адреса сокета, пославшего дейтаграмму. По этому адресу можно послать ответ.

В нашем примере функция *recvfrom()* примет следующий вид (параметр *flags* устанавливается как 0):

```
int rv = recvfrom(s, b, lstrlen(b), 0, (STRUCT SOCKADDR*)
&ad, &l);
```

При успешном считывании дейтаграммы функция *recvfrom()* возвращает количество прочитанных байтов. При ошибочном приеме дейтаграммы возвращается значение *socket_error*.

Если размер буфера, переданный в функцию *recvfrom()*, слишком мал для приема всей дейтаграммы целиком, буфер заполняется теми данными, которые в него помещаются, а оставшаяся часть дейтаграммы сбрасывается в подсистему сборки мусора и пропадает безвозвратно. В этом случае функция *recvfrom()* возвращает значение *socket error*.

```
b[rv]='\0';
cout<<b<<endl;

for (unsigned i=0;b[i];i++)
     if (i%2==0)
          if (b[i+1]!='\0'){

               tmp=b[i];
               b[i]=b[i+1];
               b[i+1]=tmp;
          }
```

Отправка данных выполняется функцией *sendto()*, имеющей следующий прототип:

```
int sendto (SOCKET s, const char FAR *buf, int len, int
```

```
flags, const struct sockaddr FAR *to, int tolen) ;
```

Первый параметр, как и раньше, – это дескриптор сокета; за ним идет указатель на буфер, содержащий пересылаемые данные, и длина этого буфера. Последние два параметра используются для указания адреса и порта сокета назначения.

Если функция *sendto()* срабатывает корректно, она возвращает количество посланных байтов, которое может отличаться от значения параметра len. В случае ошибки функция *sendto()* возвращает *socket_error*.

При попытке послать дейтаграмму большего размера, чем максимально допустимый в данной реализации *WinSock*, код ошибки будет равен *wsaemsgsize*. Максимально допустимый размер дейтаграммы можно определить путем вызова функции *WSAStartup()*.

```
        sendto(s, b, lstrlen(b), 0, (STRUCT SOCKADDR*) &ad, l);

    closesocket(s);

    WSACleanup();
}
```

Клиентская часть

Ниже приведена программа клиента.

```
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>
#include <iostream.h>

int main(void){
    char buf[100], b[100];
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD(2,2);
    err=WSAStartup(wVersionRequested, &wsaData);
    if(err != 0){return 0;}

    SOCKET s;
    s = socket(AF_INET,SOCK_DGRAM,0);
    sockaddr_in add;
    add.sin_family = AF_INET;
    add.sin_port = htons(1024);
```

В следующей строке происходит явное указание *IP*-адреса при помощи строкового представления (точечной нотации) *inet_addr*, возвращающего число в формате поля *s_addr* .

```
        add.sin_addr.s_addr = inet_addr("127.0.0.1");


        int t;
        t = sizeof(add);
        cout<<"Enter the string, please"<<endl;
        cin.getline(buf,100,'\n');

        sendto(s, buf, lstrlen(buf), 0, (struct sockaddr*) &add,
    t);

        int rv = recvfrom(s, b, lstrlen(b), 0, (struct sockaddr*)
&add, &t);
        b[rv] = '\0';
        cout<<b<<endl;

        closesocket(s);

        WSACleanup();
        return 0;
      }
```

2.5 Индивидуальные задания

Разработать приложение, реализующее архитектуру «клиент-сервер». Необходимо реализовать последовательный сервер без установления логического соединения (*UDP*). Логику взаимодействия клиента и сервера реализовать следующим образом:

1. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна 3, то удаляются все числа, которые делятся на 3. Результаты преобразований этой строки возвращаются назад клиенту.

2. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина четная, то удаляются 3 первых и 2 последних символа. Результаты преобразований этой строки возвращаются назад клиенту.

3. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен выяснить, имеются ли среди символов этой строки все буквы, входящие в слово WINDOWS. Количество вхождений символов в строку передать назад клиенту.

4. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина не-

четная, то удаляется символ, стоящий посередине строки. Преобразованная строка передается назад клиенту.

5. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен заменить в этой строке каждый второй символ @ на #. Результаты преобразований передаются назад клиенту.

6. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен заменить в этой строке символов все пробелы на символ *. Преобразованная строка передается назад клиенту.

7. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина больше 15, то из нее удаляются все цифры. Клиент получает преобразованную строку и количество удаленных цифр.

8. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина кратна 4, то удаляются все числа, делящиеся на 4. Клиент получает преобразованную строку и количество таких чисел.

9. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна 5, то подсчитывается количество скобок всех видов. Их количество посылается клиенту.

10. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна 4, то первая часть строки меняется местами со второй. Результаты преобразований возвращаются назад клиенту.

11. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если значение этой длины равно 10, то удаляются все символы от A до Z. Результаты преобразований такой строки и количество удалений возвращаются назад клиенту.

12. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина больше 15, то удаляются все символы от a до z. Преобразованная строка и количество удаленных символов возвращаются назад клиенту.

13. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен в полученной строке символов поменять местами символы на четных и нечетных позициях. Полученную строку возвратить назад клиенту.

14. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина больше 7, то выделяется подстрока в { } скобках и возвращается назад клиенту.

15. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки и, если длина больше 15, то выделяется подстрока до первого пробела и возвращается назад клиенту.

2.6 Контрольные вопросы

1. Что содержит *UDP*-сообщение помимо посылаемых данных?
2. Что называется дейтаграммой?
3. Какие возможности не предоставляет *UDP* (в отличие от *TCP*)?
4. Чем функции *sendto()* и *recvfrom()* отличаются от функций *send()* *recv()*?
5. Что происходит, если размер буфера, переданный в функцию *recvfrom*(), слишком мал для приема всей дейтаграммы целиком?
6. Приведите примеры сетевых приложений, использующих *UDP*.
7. В каких случаях применение *UDP*-протокола может быть предпочтительней, чем *TCP*?

ЛАБОРАТОРНАЯ РАБОТА №3

Создание параллельного многопоточного сервера с установлением логического соединения TCP

Цель работы: изучить методы создания серверных приложений на основе установления логического соединения *TCP*, используя алгоритм многопоточной обработки запросов.

В предыдущих лабораторных работах были показаны примеры реализации последовательных серверов как с установлением, так и без установления логического соединения. В данной лабораторной работе рассматривается пример параллельного сервера с установлением логического соединения.

Многопоточность – это специализированная форма многозадачности (*multitasking*). Что касается многозадачности, то выделяют два типа многозадачности: основанную на процессах (*process-based*) и основанную на потоках (*thread-based*). По сути, процесс (*process*) – это отдельно выполняющаяся программа. Таким образом, основанная на процессах многозадачность – средство, позволяющее вашему компьютеру выполнять несколько программ одновременно.

Отличия основанной на процессах и многопоточной многозадачности можно сформулировать следующим образом: первая поддерживает одновременное выполнение нескольких программ, а вторая имеет дело с одновременным выполнением разных фрагментов одной и той же программы.

С помощью процессов можно организовать параллельное выполнение программ. Для этого процессы клонируются с помощью вызовов функции *fork()* или функции *exec()*, а затем между ними (процессами) организуется взаимодействие средствами *IPC*. Это довольно дорогостоящий с точки зрения ресурсов процесс.

С другой стороны, для организации параллельного выполнения и взаимодействия части программы можно использовать механизм многопоточности. Основной единицей здесь является поток. Рассмотрим этот механизм подробнее.

3.1 Потоки

Последовательная реализация сервера, о которой речь шла в предыдущих лабораторных работах, может оказаться неподходящей, поскольку клиенты будут вынуждены ждать завершения обработки всех предыдущих запросов на установление соединения. Если клиент решит передать большие объемы данных (например, несколько мегабайт), последовательный сервер отложит обслуживание всех других клиентов до тех пор, пока не выполнит этот запрос.

Параллельная реализация сервера дает возможность обойтись без продолжительных задержек, так как не позволяет одному клиенту захватить все ресурсы. Вместо этого параллельный сервер поддерживает обмен данными сразу с несколькими клиентами для того, чтобы их запросы обрабатывались одновременно. Поэтому, с точки зрения клиента, параллельный сервер обеспечивает лучшее наблюдаемое время отклика по сравнению с последовательным сервером.

Распараллеливание обработки на сервере достигается созданием отдельного потока для обработки запросов одного клиента или отдельного однопотокового процесса для обработки запросов одного клиента.

Поток (*thread*) – это управляемая единица исполняемого кода. У всех процессов обязательно есть один поток, но их может быть и больше. Это означает, что в одной программе могут выполняться несколько задач одновременно. Примером может служить текстовый редактор, в котором возможны одновременные форматирование текста и печать, осуществляемые в отдельных потоках.

3.2 Преимущества многопоточности

Если операционная система поддерживает концепции потоков в рамках одного процесса, она называется многопоточной. Многопоточные приложения имеют ряд преимуществ:

а) улучшенная реакция приложения – любая программа, содержащая много не зависящих друг от друга действий, может быть перепроектирована так, чтобы каждое действие выполнялось в отдельном потоке. Например, пользователь многопоточного интерфейса не должен ждать завершения одной задачи, чтобы начать выполнение другой;

б) более эффективное использование мультипроцессирования – как правило, приложения, реализующие параллелизм через потоки, не должны учитывать число доступных процессоров. Производительность приложения равномерно увеличивается при наличии дополнительных процессоров. Численные алгоритмы и приложения с высокой степенью параллелизма, например перемножение матриц, могут выполняться намного быстрее;

в) улучшенная структура программы – некоторые программы более эффективно представляются в виде нескольких независимых или полуавтономных единиц, чем в виде единой монолитной программы. Многопоточные программы легче адаптировать к изменениям требований пользователя;

г) эффективное использование ресурсов системы – программы, использующие два или более процессов, которые имеют доступ к общим данным через разделяемую память, содержат более одного потока управления. При этом каждый процесс имеет полное адресное пространство и состояние в операционной системе. Стоимость создания и поддержания большого количества служебной информации делает каждый процесс более затратным, чем поток. Кроме того, разделение работы между процессами может потребовать от программиста значительных усилий, чтобы обеспечить связь между потоками в различных процессах или синхронизировать их действия.

3.3 Преимущества и недостатки многопотоковых процессов

Многопотоковые процессы обладают двумя основными преимуществами по сравнению с однопотоковыми процессами: более высокая эффективность и разделяемая память. Повышение эффективности связано с уменьшением издержек на переключение контекста. Переключателем контекста называются действия, выполняемые операционной системой при передаче ресурсов процессора от одного потока выполнения к другому. При переключении с одного потока на другой операционная система должна сохранить в памяти состояние предыдущего потока (например, значение регистров) и восстановить состояние следующего потока. Потоки в одном и том же процессе разделяют значительную часть информации о состоянии процесса, поэтому операционной системе приходится выполнять меньший объем работы по сохранению и восстановлению состояния. Вследствие этого переключение с одного потока на другой в одном и том же процессе происходит быстрее по сравнению с переключением между двумя потоками в разных процессах. В частности, поскольку потоки одного и того же процесса разделяют адресное пространство памяти, то переключение между потоками процесса означает, что операционная система не должна менять отображение виртуальной памяти на физическую.

Второе преимущество потоков, т.е. разделяемая память, вероятно, является для программистов еще более важным, чем повышение эффективности. Потоки упрощают разработку параллельных серверов, в которых все копии сервера должны взаимодействовать друг с другом или обращаться к разделяемым элементам данных. Кроме того, потоки упрощают разработку систем контроля и управления. В частности, поскольку ведомые потоки в сервере совместно используют память, они могут записывать в глобальную память статистическую информацию, что позволяет контролирующему потоку формировать отчеты об активности ведомых потоков сервера для системного администратора.

Хотя потоки имеют свои преимущества над однопотоковыми процессами, они не лишены также определенных недостатков. Один из наиболее важных недостатков связан с тем, что потоки не только разделяют память, но и имеют общее состояние процесса, поэтому действия, выполненные одним потоком, могут повлиять на другие потоки в том же процессе. Например, если два потока попытаются одновременно обратиться к одной и той же переменной, они могут помешать друг другу.

API-интерфейс потоков предоставляет функции, которые могут использоваться потоками для координации работы. Однако многие библиотечные функции, возвращающие указатели на статические элементы данных, не являются безопасными с точки зрения потоков, а это означает, что результаты вызова таких функций могут оказаться непредсказуемыми.

Еще один недостаток потоков (и отличие однопотокового процессора от многопотокового) связан с отсутствием надежности. Если одна из параллельно работающих копий однопотокового сервера вызовет серьезную ошибку (например, в ней будет выполнена ссылка на недопустимую область памяти), то

операционная система завершит только тот процесс, который вызвал ошибку. С другой стороны, если серьезная ошибка будет вызвана одним из потоков многопотокового сервера, то операционная система завершит весь процесс.

3.4. Алгоритм работы параллельного (многопоточного) сервера с установлением логического соединения

Приведем обобщенный алгоритм работы параллельного сервера с установлением логического соединения:

1. Ведущий поток. Создать сокет и выполнить его привязку к локальному адресу. Оставить сокет неподключенным.
2. Ведущий поток. Перевести сокет в пассивный режим, подготовив его для использования сервером.
3. Ведущий поток. Вызывать в цикле функцию *accept* для получения очередного запроса от клиента и создавать новый ведомый поток или процесс для формирования ответа.
4. Ведомый поток. Работа потока начинается с получения доступа к соединению, полученному от ведущего потока (т.е. к сокету соединения).
5. Ведомый поток. Выполнять обмен данными с клиентом через соединение: принимать запрос (запросы) и передавать ответ (ответы).
6. Ведомый поток. Закрыть соединение и завершить работу. Ведомый поток завершает свою работу после обработки всех запросов от одного клиента.

Сервер функционирует неопределенно долгое время, ожидая поступления новых запросов на установление соединения от клиентов. Его ведущий поток при подключении клиента создает новый ведомый поток для обработки запросов каждого нового соединения и предоставляет каждому ведомому потоку возможность взять на себя весь обмен данными с клиентом. На рисунке 4 приведена схема организации потоков параллельного сервера с установлением логического соединения.

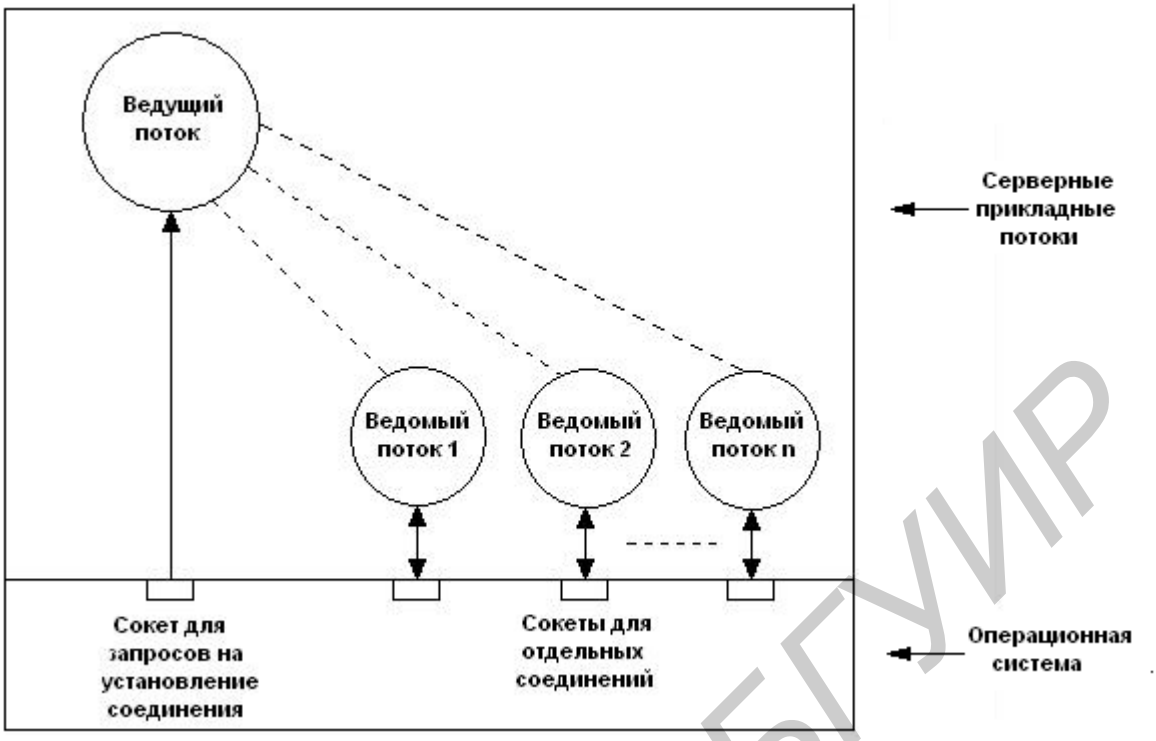

Рисунок 4 – Схема организации потоков параллельного сервера с установлением логического соединения

3.5. Методические указания по созданию параллельного многопоточного сервера с установлением логического соединения TCP

Осуществить взаимодействие клиента и сервера на основе протокола *TCP/IP*. Реализовать параллельное соединение с использованием многопоточности. Функциональные возможности клиента реализовать следующим образом: клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Функциональные возможности сервера реализовать следующим образом: сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна четырем, то первая часть строки меняется местами со второй. Результаты преобразований возвращаются назад клиенту.

Серверная часть

```
#include<stdio.h>
#include<iostream.h>
#include <winsock2.h>
```

Функция *CreateThread()* создает поток, который выполняется в пределах адресного пространства вызова процесса и имеет следующий прототип:

```
HANDLE Create Thread (
LPSECURITY_ATTRIBUTES lpThreadAttributes,    // указатель на
атрибуты безопасности
```

```
    DWORD dwStackSize, // размер стека начального потока
    LPTHREAD_START_ROUTINE lpStartAddress, // указатель на функ-
цию потока
    LPVOID lpParameter,  // аргумент для нового потока
    DWORD dwCreationFlags, // создание флагов
    LPDWORD lpThreadId // указатель на ID поток для его получения
     );
```

Рассмотрим каждый параметр подробнее:

lpThreadAttributes – представляет собой указатель на структуру *SECURITY_ATTRIBUTES*, который определяет, может ли дескриптор потока быть унаследован дочерними процессами. Если *lpThreadAttributes* принимает значение *NULL*, дескриптор потока не может быть унаследован.

Для *MS Windows NT*: член структуры *lpSecurityDescriptor* определяет дескриптор безопасности для нового потока. Если *lpThreadAttributes* принимает значение *NULL*, то поток получает дескриптор безопасности по умолчанию.

dwStackSize – эта величина определяет начальный размер стека в байтах. Система округляет это значение до ближайшей страницы. Если это значение равно нулю или меньше размера стека по умолчанию, то используется тот же размер, что и при вызове потока. Стек освобожден в том случае, когда поток завершается.

lpStartAddress – указатель на определенную прикладную функцию типа *LPTHREAD_START_ROUTINE* для выполнения ее потоком и представления начального адреса потока.

lpParameter – определяет единственную 32-битовую величину параметра, переданную в поток.

dwCreationFlags – используется для определения дополнительных флагов, которые управляют созданием потока. Если флаг *CREATE_SUSPENDED* определен, поток создан в приостановленном состоянии и не будет работать, пока функция *ResumeThread()* не будет вызвана. Если эта нулевая величина, то поток выполняется немедленно после его создания. В то же время никакие другие величины не предусмотрены.

lpThreadId – указатель на 32-битовую переменную, которая получает идентификатор потока.

Для *MS Windows NT***:** Если этот параметр принимает значение *NULL*, то идентификатор потока не возвращается.

Для *MS Windows 95* и *MS Windows 98*: Этот параметр может не принимать значение *NULL*.

Если функция *CreateThread()* успешно выполняется, то возвращаемое значение есть указатель на новый поток. В случае невыполнения функции возвращаемое значение принимает значение *NULL*. Для получения большей информации об ошибке обратимся к функции *GetLastError()*.

Примечания –

- Новый поток управления создается макросом *THREAD_ALL_ACCESS* для нового потока. Если дескриптор безопасности не предусмотрен, то управление может быть использовано в любой функции, которая требует объектного управления потоком. Когда дескриптор безопасности предусмотрен, то контроль доступа выполняется во всех последующих использованиях дескриптора прежде, чем доступ будет предоставлен. Если контроль доступа запрещает доступ, запрашиваемый процесс не может использовать дескриптор для получения доступа к потоку.
- Выполнение потока начинается в функции, определенной параметром *lpStartAddress*. Если эта функция возвращает значение типа *DWORD*, то оно используется для завершения потока неявным вызовом функции *ExitThread()*, которая будет описана ниже. Используйте функцию *GetExitCodeThread()*, чтобы получать возвращаемое значение потока.
- Функция *CreateThread()* выполняется, даже если указатель *lpStartAddress* указывает на данные, код, или недоступен. Если начальный адрес недействителен во время работы потока, срабатывает исключение и поток завершается. Завершение потока из-за неправильного начального адреса интерпретируется как аварийный выход процесса потока.

В нашей программе основной алгоритм решения задачи находится в функции *ThreadFunc()*, которую определим следующим образом:

```
DWORD WINAPI ThreadFunc(LPVOID client_socket){
     SOCKET s2=((SOCKET *) client_socket)[0];
     char buf[100];
     char buf1[100];
     while(recv(s2,buf,sizeof(buf),0)){
          int k, j=0;
          k=strlen(buf)-1;

          if(k%4==0){
               for(int i=k/2; i<k; i++){
                    buf1[i]=buf[j];
                    j++;
               }

               for(i=0; i<k/2; i++){
                    buf1[i]=buf[j];
                    j++;
               }

               buf1[k]='\0';
               strcpy(buf,buf1);
          }
          cout<<buf<<endl;
          send(s2,buf,100,0);
```

```
    }
    closesocket(s2);
    return 0;
}
```

При вызове функции *ThreadFunc()* в основной программе (main-функции) передается дескриптор сокета.

```
int numcl=0;

void print(){
    if (numcl) printf("%d client connected\n",numcl);
    else printf("No clients connected\n");
}

void main()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD( 2, 2 );
    err = WSAStartup( wVersionRequested, &wsaData );
    if ( err != 0 ){return;}

    SOCKET s=socket(AF_INET,SOCK_STREAM,0);
    sockaddr_in local_addr;
    local_addr.sin_family=AF_INET;
    local_addr.sin_port=htons(1280);
    local_addr.sin_addr.s_addr=0;
    bind(s,(sockaddr *) &local_addr,sizeof(local_addr));
    int c=listen(s,5);
    cout<<"Server receive ready"<<endl;
    cout<<endl;
    // извлекаем сообщение из очереди
    SOCKET client_socket;    // сокет для клиента
    sockaddr_in client_addr; // адрес клиента (заполняется
//системой)
    int client_addr_size=sizeof(client_addr);
    // цикл извлечения запросов на подключение из очереди
    while((client_socket=accept(s,(sockaddr   *)&client_addr,
&client_addr_size))){
        numcl++;
        print();
        // вызов нового потока для обслуживания клиента
        DWORD thID;// thID идентификатор типа DWORD
        CreateThread(NULL,NULL,ThreadFunc,
        &client_socket,NULL,&thID)
    }
}
```

По окончании работы функция *CreateThread()* закрывает поток, инициализирует используемые указатели значением *NULL*. Существует специальная функция *Exit Thread()*, выполняющая аналогичные действия. Ее прототип:

```
VOID ExitThread( DWORD dwExitCode  );
```

Параметр *dwExitCode* определяет выходной код для вызова потока. Данная функция не возвращает никакого значения.

Примечания –

- Когда функция *ExitThread()* явно вызвана, текущий стек потока освобожден и поток завершается.
- Если поток является последним в процессе, когда эта функция вызвана, то процесс потока также завершается.
- Завершение потока не обязательно удаляет его объект из операционной системы. Объект потока удален, когда закрывается последний дескриптор потока.

Клиентская часть

Во многом дублирует клиент-приложения предыдущих лабораторных работ. Согласно требованиям условия задачи клиентская часть имеет следующий вид:

```
#include<stdio.h>
#include<iostream.h>
#include<winsock2.h>

void main()
{
     WORD wVersionRequested;
     WSADATA wsaData;
     int err;
     wVersionRequested = MAKEWORD( 2, 2 );
     err=WSAStartup( wVersionRequested, &wsaData );
     if ( err != 0 ){return;}

while (true){
     SOCKET s=socket(AF_INET,SOCK_STREAM,0);
     // указание адреса и порта сервера
     sockaddr_in dest_addr;
     dest_addr.sin_family=AF_INET;
     dest_addr.sin_port=htons(1280);
     dest_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
     connect(s,(sockaddr *)&dest_addr,sizeof(dest_addr));

     char buf[100];
```

```
            cout<<"Enter the string:"<<endl;
            fgets(buf,sizeof(buf),stdin);
            send(s,buf,100,0);

            if (recv(s,buf,sizeof(buf),0)!=0){
                 cout<<"Poluchenaya stroka:"<<endl<<buf<<endl;
            }
      closesocket(s);
}
WSACleanup();
}
```

3.6 Индивидуальные задания

Разработать приложение, реализующее архитектуру «клиент-сервер». Для этого необходимо реализовать параллельный многопоточный сервер с установлением логического соединения (TCP). Логику взаимодействия клиента и сервера реализовать так, как указано в варианте индивидуального задания. Предусмотреть возможность просмотра, добавления, редактирования, удаления информации клиентом на сервере.

1. На сервере хранится список студентов. Каждая запись списка содержит следующую информацию о студенте:

– ФИО студента;
– номер группы;
– размер стипендии;
– оценки по N предметам.

Таких записей должно быть не менее пяти.

Клиент вводит с клавиатуры букву алфавита, по которой он хотел бы посмотреть информацию о студентах, и посылает ее на сервер. Назад он получает список только тех студентов, фамилии которых начинаются на эту букву.

2. На сервере хранится список студентов. Каждая запись списка содержит следующую информацию о студенте:

– ФИО студента;
– номер группы;
– размер стипендии;
– оценки по N предметам.

Таких записей должно быть не менее 5-ти.

Клиент посылает на сервер средний балл студента, по которому он хочет получить информацию о студентах. Назад он получает список только тех студентов, средний балл которых больше заданного.

3. На сервере хранится список о студентах. Каждая запись списка содержит следующую информацию о студенте:

– ФИО студента;
– номер группы;
– размер стипендии;
– оценки по N предметам.

Таких записей должно быть не менее 5-ти.

По запросу клиента он получает от сервера список только тех студентов, которые не имеют оценки 3.

4. На сервере хранится список сотрудников фирмы. Каждая запись списка содержит следующую информацию о сотрудниках:
– ФИО сотрудника;
– табельный номер;
– количество отработанных часов за месяц;
– почасовой тариф.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер величину заработной платы, по которой он хочет получить информацию о сотрудниках. Назад он получает список только тех сотрудников, заработная плата которых меньше указанной.

5. На сервере хранится информация (список) о комплектующих деталях. Каждая запись списка содержит следующую информацию о комплектующем:
– завод-поставщик;
– стоимость;
– дата поставки;

Таких записей должно быть не менее семи.

Клиент посылает на сервер дату поставки. Назад он получает список комплектующих, поставленных именно на эту дату, и стоимость каждого возвращаемого комплектующего должна превосходить минимальную во всем списке.

6. На сервере хранится список автобусных рейсов. Каждая запись списка содержит следующую информацию о рейсе:
– номер рейса;
– тип автобуса;
– цена билета;
– пункт назначения.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер пункт назначения. Назад он получает список рейсов, позволяющих добраться до заданного пункта.

7. На сервере хранится список игроков. Каждая запись списка содержит следующую информацию об игроках:
– ФИО игрока;
– игровой номер;
– возраст;
– рост;
– вес.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер запрос и получает информацию о самом молодом игроке.

8. На сервере хранится список разговоров на междугородной АТС. Каждая запись списка содержит следующую информацию о разговорах:

- дату разговора;
- код и название города;
- продолжительность разговора;
- тариф;
- номер телефона в этом городе;
- номер телефона абонента.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер название города. Назад он получает суммарное время разговора с указанным городом.

9. На сервере хранится список товаров, имеющихся на складе. Каждая запись списка содержит следующую информацию о товарах:

- страна-изготовитель;
- фирма-изготовитель;
- наименование товара;
- количество единиц товара.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер название страны-изготовителя. Назад он получает товары и их данные для указанной страны.

10. На сервере хранится список книг, хранящихся в библиотеке. Каждая запись списка содержит следующую информацию о книгах:

- регистрационный номер книги,
- автор;
- название;
- год издания;
- издательство;
- количество страниц.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер фамилию интересующего его автора. Назад он получает список книг указанного автора.

11. На сервере хранится список деталей. Каждая запись списка содержит следующую информацию о деталях:

- наименование детали;
- количество деталей;
- номер цеха, где они изготовлены.

Таких записей должно быть не менее семи.

Клиент посылает на сервер наименование детали. Назад он получает общее количество изделий указанного наименования.

12. На сервере хранится список товаров, имеющихся на складе. Каждая запись списка содержит следующую информацию о товарах:

– страна-изготовитель;
– фирма-изготовитель;
– наименование товара;
– количество единиц товара.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер наименование товара. Назад он получает количество единиц этого товара.

13. На сервере хранится каталог туристических предложений. Каждая запись каталога содержит:

– название тура;
– стоимость;
– продолжительность;
– вид транспорта.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер предполагаемую стоимость тура. Назад он получает названия тех туров, стоимость которых не выше посланной.

14. На сервере хранится список преподавателей. Каждая запись списка содержит следующие данные:

– ФИО;
– ученая степень;
– стаж работы;
– предмет.

Клиент посылает на сервер название предмета. Назад он получает ФИО тех преподавателей этого предмета, стаж работы которых не менее 5 лет.

15. На сервере хранится информация об участниках соревнований по спортивным танцам. Она включает следующие данные:

– название танцевальной пары;
– город;
– оценка жюри;
– оценка зрителей;
– возрастная группа.

Клиент посылает на сервер возраст участников. Назад он получает название той танцевальной пары, суммарная оценка которой в данной возрастной группе максимальна.

3.7 Контрольные вопросы

1. Что такое параллельное соединение? Особенности параллельного соединения.

2. В чём заключается отличие параллельного соединения от последовательного?

3. Назовите преимущества многопотоковых процессов по сравнению с однопотоковыми процессами.

4. С чем связано повышение эффективности многопотоковых процессов?

5. Назовите недостатки многопотокового процесса по сравнению с однопотоковым.

6. Какие функции служат для создания потоков?

ЛАБОРАТОРНАЯ РАБОТА №4

Создание параллельного многопроцессного сервера с установлением логического соединения TCP

Цель работы: изучить методы создания серверов с установлением логического соединения *TCP*, используя алгоритм параллельной обработки запросов, основанный на многопроцессности.

В настоящей лабораторной работе приведен пример параллельного сервера с установлением логического соединения. Для обеспечения параллельного формирования ответов на запросы клиентов сервер опирается на поддержку параллельного выполнения процессов операционной системой. Рассматривается реализация, в которой используется несколько однопотоковых процессов.

4.1 Основные сведения о процессах

В некоторых операционных системах (например ОС *Linux*) управление процессами является ключевой технологией при разработке многих программ.

Процесс – это находящаяся в состоянии выполнения программа, включающая ее среду выполнения.

Так, например, *Linux* – это многозадачная система, в которой одновременно могут выполняться несколько программ (процессов, задач). Каждый процесс имеет собственное виртуальное адресное пространство. Это необходимо, чтобы гарантировать, что ни один из процессов не будет подвержен помехам или влиянию со стороны других. Отдельные процессы получают доступ к центральному процессору (ЦП) по очереди. Планировщик процессов решает, как долго и в какой последовательности процессы будут занимать ЦП. При этом создается впечатление, что процессы протекают действительно параллельно.

В ОС *Linux* реализована вытесняющая многозадачность. Это значит, что система сама решает, как долго конкретный процесс может использовать ЦП и когда наступит очередь обработки для следующего процесса.

Возможны следующие состояния процесса:

S – неактивный (спящий) процесс;

SW – процесс выгружен на устройство выгрузки;

R – активный (выполняющийся) процесс;

Z – "зомби" (процесс является завершенным, однако он не послал статус возврата в родительский процесс);

T – процесс остановлен.

4.2 Создание процессов

Создать новый процесс в ОС *Linux* можно с помощью системного вызова функции *fork()*. Синтаксис вызова следующий:

```
#include <sys/types>
#include <unistd.h>
pid_t fork(void);
```

pid_t является примитивным типом данных, который определяет идентификатор процесса или группы процессов. При вызове *fork()* порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю.

Процесс-потомок наследует следующие признаки родителя:

- сегменты кода, данных и стека программы;
- таблицу файлов, в которой находятся состояния флагов дескрипторов файла, указывающие, читается ли файл или пишется. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения;
- рабочий и корневой каталоги;
- реальный и эффективный номер пользователя и номер группы;
- приоритеты процесса (администратор может изменить их через nice);
- контрольный терминал;
- маска сигналов;
- ограничения по ресурсам;
- сведения о среде выполнения;
- разделяемые сегменты памяти.

Потомок не получает от родителя следующие признаки:

- идентификатор процесса (*PID*, *PPID*);
- израсходованное время ЦП (оно обнуляется);
- сигналы процесса-родителя, требующие ответа;
- блокированные файлы (*record locking*).

При вызове *fork()* возникают два полностью идентичных процесса. Весь код после *fork()* выполняется дважды: как в процессе-потомке, так и в процессе-родителе.

Процесс-потомок и процесс-родитель получают разные коды возврата после вызова *fork()*. Процесс-родитель получает идентификатор (*PID*) потомка. Если это значение будет отрицательным, значит, при порождении процесса произошла ошибка. Процесс-потомок получает в качестве кода возврата значение 0, если вызов *fork()* произошел успешно.

Таким образом, можно проверить, был ли создан новый процесс.

```
switch(ret=fork()){
case -1: /{*}при вызове fork() возникла ошибка{*}/
case 0 : /{*}это код потомка{*}/
default : /{*}это код родительского процесса{*}/
}
```

4.3 Различие между процессами и потоками

Изучив сущность и основные методы создания и работы с процессами и потоками, приведем основные различия между ними:

– поток представляет собой облегченную версию процесса. Чтобы понять, в чем состоит его особенность, необходимо привести основные характеристики процесса;

– процесс владеет определенными ресурсами. Он размещен в некотором виртуальном адресном пространстве, содержащем образ процесса. Кроме того, процесс управляет другими ресурсами (файлы, устройства ввода – вывода и т. д.);

– процесс подвержен диспетчеризации. Он определяет порядок выполнения одной или нескольких программ, при этом выполнение может перекрываться с другими процессами. Каждый процесс имеет состояние выполнения и приоритет диспетчеризации.

Если рассматривать эти характеристики независимо друг от друга (как это принято в современной теории операционных систем), то:

- владелец ресурса обычно называется процессом или задачей. Ему присущи:
    - виртуальное адресное пространство;
    - индивидуальный доступ к процессору, другим процессам, файлам, и ресурсам ввода – вывода.
- модуль для диспетчеризации обычно называется потоком или облегченным процессом. Ему присущи:
    - состояние выполнения (активное, готовность и т.д.);
    - сохранение контекста потока в неактивном состоянии;
    - стек выполнения и некоторая статическая память для локальных переменных;
    - доступ к пространству памяти и ресурсам своего процесса.

Все потоки процесса разделяют общие ресурсы. Изменения, вызванные одним потоком, становятся немедленно доступны другим потокам.

При корректной реализации потоки имеют определенные преимущества над процессами. Им требуется меньше времени:

- для создания нового потока, поскольку создаваемый поток использует адресное пространство текущего процесса;
- для завершения потока;
- для переключения между двумя потоками в пределах того же самого процесса.

Потокам требуется также меньше коммуникационных расходов, поскольку потоки разделяют все ресурсы, и в частности, адресное пространство. Данные, продуцируемые одним из потоков, немедленно становятся доступными всем другим потокам.

4.4 Алгоритм работы параллельного (многопроцессного) сервера с установлением логического соединения

Приведем обобщенный алгоритм работы параллельного (многопроцессного) сервера с установлением логического соединения:

1. Ведущий процесс. Создать сокет и выполнить его привязку к локальному адресу. Оставить сокет неподключенным.

2. Ведущий процесс. Перевести сокет в пассивный режим, подготовив его для использования сервером.

3. Ведущий процесс. Вызывать в цикле функцию *accept* для получения очередного запроса от клиента и создавать новый ведомый процесс через *fork*.

4. Ведомый процесс. Работа процесса начинается с получения доступа к соединению, полученному от ведущего процесса (т.е. к сокету соединения).

5. Ведомый процесс. Выполнять обмен данными с клиентом через соединение: принимать запрос (запросы) и передавать ответ (ответы).

6. Ведомый процесс. Закрыть соединение и завершить работу. Ведомый процесс завершает свою работу после обработки всех запросов от одного клиента.

На рисунке 5 показана схема организации процессов параллельного сервера с установлением логического соединения, в котором используются однопотоковые процессы. Как показано на этом рисунке, ведущий процесс непосредственно не взаимодействует с клиентами. По существу, в его задачу входит просто ожидание поступления очередного запроса на установление соединения через общепринятый порт. После поступления такого запроса система возвращает новый дескриптор сокета, предназначенный для использования с этим соединением. Ведущий процесс создает ведомый процесс для обслуживания соединения и предоставляет возможность ведомому процессу работать параллельно с ним. В любое время в сервере функционирует один ведущий процесс, а число ведомых процессов может составлять от нуля и более. Ведущий процесс сервера принимает каждый входящий запрос на установление соединения и создает ведомый процесс для его обслуживания.

Для получения очередного запроса на установление соединения из общепринятого порта используется блокирующий вызов функции *accept*, что позволяет предотвратить бесполезное расходование ресурсов процессора во время ожидания очередных запросов на установление соединения в ведущем процессе сервера. Поэтому в отличие от последовательного сервера ведущий процесс сервера в параллельном сервере основную часть времени проводит в состоянии, заблокированном в вызове функции *accept*. После поступления запроса на установление соединения выполняется возврат управления функцией *accept*, а ведущий процесс получает возможность продолжить выполнение. Ведущий процесс создает ведомый процесс для обработки запроса и снова вызывает функцию *accept*. В этом вызове ведущий процесс в очередной раз блокируется до поступления еще одного запроса на установление соединения.

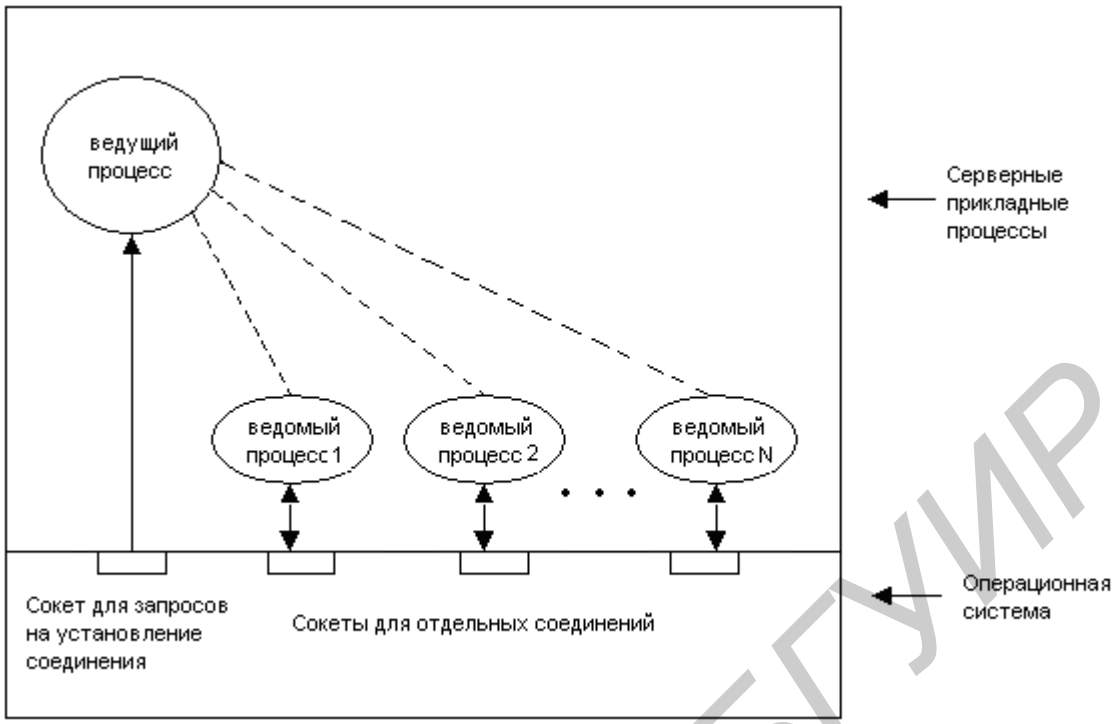

Рисунок 5 – Схема организации процессов параллельного сервера с установлением логического соединения, в котором используются однопотоковые процессы

4.5 Методические указания по созданию параллельного сервера с установлением логического соединения TCP, использующего отдельный процесс для обработки запросов клиента

Рассмотрим последовательность действий по созданию такого сервера на конкретном примере.

Осуществить взаимодействие клиента и сервера на основе протокола TCP. Реализовать параллельное соединение с использованием многопроцессности. На сервере хранится список сотрудников фирмы. Каждая запись списка содержит следующую информацию о сотрудниках:

- ФИО сотрудника;
- табельный номер;
- доход в месяц;
- ставка налогов, отчисляемых в бюджет.

Таких записей должно быть не менее пяти.

Клиент посылает на сервер количество месяцев и символ алфавита (первая буква фамилии сотрудника) и получает назад общую сумму отчислений налогов в бюджет, уплачиваемых сотрудниками, фамилии которых начинаются с этого символа, в течение этих месяцев. Предусмотреть возможность редактирования записей списка клиентом.

Решение этой задачи подразумевает использование компилятора языка *C++ gcc*, поставляемого вместе с эмулятором операционной системы *Unix – Cygwin*.

Для реализации поставленной задачи, как и ранее, создадим два проекта, для отладки и компиляции которых используем оболочку *Cygwin*.

Серверная часть

```
// Подключение необходимых библиотек для работы с пакетом
//Cygwin
#include <sys/types.h>
#include<sys/socket.h>
#include<sys/signal.h>
#include<sys/wait.h>
#include<sys/resource.h>
#include<netinet/in.h>

#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>


struct Employee{
     char name[35];
     char number[10];
     char income[10];
     char tax[4];
} em[5];

// процедура для обслуживания соединения
int Func(int newS){
     long int i,num,t, mon, doh, nal;
     float sum;
     int m;
     char p,p1,s;
     char buf[256],b[256];
     while (true){
          recv(newS,buf,sizeof(buf),0);
          p=buf[0];
          switch(p){
          case '1':
               buf[0]='\0';
               sum=0;
               recv(newS,buf,sizeof(buf),0);
               mon=atoi(buf);
               recv(newS,buf,sizeof(buf),0);
               s=buf[0];
               for (i=1;i<=5;i++)
                    if (em[i].name[0]==s){
```

```
                    nal=atoi(em[i].tax);
                    doh=atoi(em[i].income);
                    printf("mon %d\n",mon);
                    sum=sum+(nal*doh*mon)/100.0;
               }
          int* decpt,*sgn;
          printf("%f\n",sum);
          strcpy(buf,fcvt(sum,3,decpt,sgn));
          send(newS,buf,sizeof(buf),0);
          puts(buf);
          break;
     case '2':
          recv(newS,buf,sizeof(buf),0);// Номер
          num=atoi(buf);
          printf("%d\n",num);
          recv(newS,buf,sizeof(buf),0);
          p1=buf[0];
          switch(p1){
               case '1':
                    recv(newS,buf,sizeof(buf),0);// Имя
                    strcpy(em[num].name,buf);
                    break;
               case '2':
                    recv(newS,buf,sizeof(buf),0);//Таб.номер
                    strcpy(em[num].number,buf);
                    break;
               case '3':
                    recv(newS,buf,sizeof(buf),0);//Доход
                    strcpy(em[num].income,buf);
                    break;
               case '4':
               recv(newS,buf,sizeof(buf),0);//Ставка налога
                              strcpy(em[num].tax,buf);

               }
               break;
          case '3':
               for (i=1;i<=5;i++){
                    buf[0]='\0';
               strcat(buf,em[i].name);strcat(buf," ");
               strcat(buf,em[i].number);strcat(buf," ");
               strcat(buf,em[i].income);strcat(buf," ");
               strcat(buf,em[i].tax);strcat(buf,"\n");
          send(newS,buf,sizeof(buf),0);
               }
               break;
          case '4':
               exit(0);
          }
     }
}
```

Поскольку в параллельных серверах для динамического создания процессов используется функция *fork()* (см. ниже), они являются потенциальным источником проблемы не полностью завершившихся процессов (процессов, информация о которых остается в системных таблицах). В системе *Linux* эта проблема решается путем передачи специального сигнала родительскому процессу после завершения работы каждого дочернего процесса. Завершившийся процесс остается в виде так называемого процесса-зомби до тех пор, пока родительским процессом не будет выполнен системный вызов *wait3*. Для полного завершения дочернего процесса (т.е. для уничтожения процесса-зомби) в рассматриваемом примере перехватывается сигнал завершения дочернего процесса и выполняется функция обработки этого сигнала. Операционной системе дается указание о том, что для ведущего процесса сервера при получении каждого сигнала о завершении работы дочернего процесса (сигнал *SIGCHLD*) должна быть выполнена функция *reaper()* в виде следующего вызова, который в нашей программе осуществляется в *main*:

```
signal (SIGCHLD, reaper);
```

После вызова функции *signal()* система автоматически вызывает функцию *reaper()* при получении процессом сервера каждого сигнала *SIGCHLD*.

Функция *reaper()* вызывает системную функцию *wait3()* для полного завершения дочернего процесса, закончившего свою работу. Функция *wait3()* остается заблокированной до тех пор, пока не произойдет завершение работы одного или нескольких дочерних процесса (по любой причине). Эта функция возвращает значение структуры *status*, которую можно проанализировать для получения дополнительной информации о завершившемся процессе. Поскольку данная программа вызывает функцию *wait3()* при получении сигнала *SIGCHLD*, вызов этой функции всегда происходит только после завершения работы дочернего процесса. Для предотвращения возникновения в сервере тупиковой ситуации в случае ошибочного вызова в программе используется параметр *WNOHANG*, который указывает, что функция *wait3* не должна блокироваться в ожидании завершения какого-либо процесса. Вместо этого возврат управления после вызова происходит немедленно, даже если не произошел выход из какого-либо процесса.

```
void reaper(int sig){
     int status;
     while (wait3(&status,WNOHANG,(struct rusage*)0)>=0);
}

int main(){
          strcpy(em[1].name,"Sergeev Sergei Sergeevich");
          strcpy(em[1].number,"1");
          strcpy(em[1].income,"100000");
          strcpy(em[1].tax,"10");
```

```
            strcpy(em[2].name,"Ivanov Ivan Ivanovich");
            strcpy(em[2].number,"2");
            strcpy(em[2].income,"200000");
            strcpy(em[2].tax,"20");

         strcpy(em[3].name,"Vladimirov Vladimir Vladimirovich");
            strcpy(em[3].number,"3");
            strcpy(em[3].income,"300000");
            strcpy(em[3].tax,"30");


            strcpy(em[4].name,"Sidorov Sidor Sidorovich ");
            strcpy(em[4].number,"4");
            strcpy(em[4].income,"400000");
            strcpy(em[4].tax,"40");

            strcpy(em[5].name,"Vasilev Vasilii Vasilievich");
            strcpy(em[5].number,"5");
            strcpy(em[5].income,"500000");
            strcpy(em[5].tax,"50");

            struct sockaddr_in local;
            int s,
                newS,
                rc;

            local.sin_family=AF_INET;
            local.sin_port=htons(7500);
            local.sin_addr.s_addr=htonl(INADDR_ANY);

            s=socket(AF_INET, SOCK_STREAM,0);
            rc=bind(s,(struct sockaddr *)&local, sizeof(local));
            rc=listen(s,5);
            (void)signal(SIGCHLD,reaper);
            while(true){
                newS=accept(s,NULL,NULL);
                switch (fork()){
                case 0:
                     (void)close(s);
                     exit(Func(newS));
                default:
                     (void)close(newS);
                }
            }
            return 0;
}
```

Клиентская часть

```
#include <sys/types.h>
```

```
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

     struct sockaddr_in peer;
     int s,t,t1;
     int rc;
     char buf[256],p,p1,b[256];;

     peer.sin_family=AF_INET;
     peer.sin_port=htons(7500);
     peer.sin_addr.s_addr=inet_addr("127.0.0.1");

     s=socket(AF_INET,SOCK_STREAM,0);
     rc=connect(s,(struct sockaddr *)&peer,sizeof(peer));
     while(true){
     //Выбор пункта меню и отправка его серверу
     puts("Choose:");
     puts("\t1 - Select");
     puts("\t2 - Edit");
     puts("\t3 - View");
     puts("\t4 - Exit");
     scanf("%s",buf);
     buf[1]='\0';
     send(s,buf,sizeof(buf),0);
     p=buf[0];

     switch (p){
     case '1'://Выбрать
          puts("kol-vo months (1-12) :");scanf("%s",buf);
          send(s,buf,sizeof(buf),0);
          puts("Symbol: ");scanf("%s",buf);
          send(s,buf,sizeof(buf),0);

          printf("Sum of taxes: ");
          recv(s,buf,sizeof(buf),0);
          for (t=0;buf[t+3];t++) printf("%c",buf[t]);
          printf(".");
          for (t1=t;buf[t1];t1++) printf("%c",buf[t1]);
          printf("\n");
          break;
     case '2'://Подредактировать
     puts("What number (1-5) to edit");
     scanf("%s",buf);//Какой номер будем редактировать
     send(s,buf,sizeof(buf),0);

     puts("What field (1-4) to edit");
```

```
            puts("\t1 - Name");
            puts("\t2 - Number");
            puts("\t3 - Income");
            puts("\t4 - Tax");
            scanf("%s",buf);
            send(s,buf,sizeof(buf),0);
            p1=buf[0];
            buf[0]='\0';
            switch(p1){          //Введите новые поля
                 case '1':printf("Name: ");
                      fflush(stdin);fflush(stdout);
                     scanf("%s",b);strcat(buf,b);strcat(buf," ");
                     scanf("%s",b);strcat(buf,b);strcat(buf," ");
                    scanf("%s",b);strcat(buf,b);strcat(buf,"\0");
                      send(s,buf,sizeof(buf),0);
                      break;
                 case '2':fflush(stdin);fflush(stdout);
                      printf("Tab Number: ");scanf("%s",buf);
                      send(s,buf,sizeof(buf),0);
                      break;
                 case '3':fflush(stdin);fflush(stdout);
                      printf("Income per month: ");
                      scanf("%s",buf);
                      send(s,buf,sizeof(buf),0);
                      break;
                 case '4':fflush(stdin);fflush(stdout);
                      printf("Tax rate (%) per month: ");
                      scanf("%s",buf);
                      send(s,buf,sizeof(buf),0);
            }
            break;
       case '3'://Просмотреть 5 записей
            recv(s,buf,sizeof(buf),0);printf("%s",buf);
            recv(s,buf,sizeof(buf),0);printf("%s",buf);
            recv(s,buf,sizeof(buf),0);printf("%s",buf);
            recv(s,buf,sizeof(buf),0);printf("%s",buf);
            recv(s,buf,sizeof(buf),0);printf("%s",buf);
            break;
       case '4'://Выход
            exit(0);

       }
}
}
```

Как показано в этом примере, вызовы функций, обеспечивающих параллельную работу, занимают лишь небольшую часть кода. Ведущий процесс сервера начинает свое выполнение в главной процедуре.

При каждом проходе по циклу ведущий сервер вызывает функцию *accept()* для перехода в состояние ожидания запроса на установление соединения от клиента. Как и в последовательном сервере, этот вызов блокируется до поступления запроса. После получения сервером *TCP* запроса на установление соединения операционная система создает сокет для нового соединения и вызов функции accept() возвращает дескриптор этого сокета.

После возврата управления из функции *accept()* ведущий процесс сервера создает ведомый процесс для обслуживания соединения. Для этого ведущий процесс вызывает функцию *fork()*, чтобы разделиться на два процесса. Поток во вновь созданном дочернем процессе вначале закрывает сокет ведущего процесса, а затем вызывает функцию *Func()* для обслуживания соединения. Поток в родительском процессе закрывает сокет, который был создан для обслуживания нового соединения, и продолжает выполнение бесконечного цикла. При следующем проходе по циклу ведущий процесс после вызова функции *accept()* снова переходит в состояние ожидания очередных запросов на установление соединения. Следует отметить, что и первоначальный, и новые процессы имеют доступ к открытым сокетам после вызова функции *fork()*, и они оба должны закрыть один из этих сокетов, после чего система освобождает связанный с ним ресурс. Поэтому этот сокет закрывается только в ведущем процессе после вызова функции *close()* потоком ведущего процесса для закрытия сокета нового соединения. Аналогичным образом, когда поток в ведомом процессе вызывает функцию *close()* для закрытия сокета ведущего процесса, этот сокет закрывается только в ведомом процессе. Ведомый процесс продолжает получать доступ к сокету нового соединения до тех пор, пока не завершит свою работу, а ведущий сервер продолжает иметь доступ к сокету, который соответствует общепринятому порту.

После закрытия сокета ведущего процесса ведомый процесс вызывает процедуру *Func()*. После возврата управления процедурой *Func()* ведомый процесс использует возвращенное значение в качестве параметра вызова функции *exit*. Система *Linux* интерпретирует вызов функции *exit* как требование завершить процесс и использует параметр вызова этой функции как код завершения процесса. В соответствии с общепринятым соглашением в процессе для обозначения нормального завершения используется код завершения нуль. После прекращения работы ведомого процесса операционная система автоматически закрывает все его открытые дескрипторы, в том числе дескриптор соединения *TCP*.

4.6 Индивидуальные задания

Разработать приложение, реализующее архитектуру «клиент-сервер». Необходимо реализовать параллельный сервер с установлением логического соединения (*TCP*), использующий отдельные процессы для обработки запросов клиентов. Логику взаимодействия клиента и сервера реализовать так, как указано в индивидуальном задании.

1. Сервер хранит информацию о домашней библиотеке. Клиент имеет возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

2. Сервер хранит информацию из записной книжки. Клиент имеет возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

3. Сервер хранит информацию о студенческой группе. Клиент имеет возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

4. Сервер хранит информацию о некоторой коллекции компакт-дисков. Клиент имеет возможность просмотра, добавления, удаления информации, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

5. Сервер хранит информацию о лицевых счетах вкладчиков банковского учреждения. Запись о каждом вкладчике состоит из информации: фамилия, дата операции, сумма вклада, общая сумма. Клиент имеет возможность пополнения и изъятия данных о вкладчиках, ведения текущих счетов, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

6. Сервер хранит информацию о персонале: фамилия, должность, зарплата, дата рождения. Клиент имеет возможность добавления, изменения, удаления информации, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

7. Сервер хранит информацию о плоских геометрических фигурах: прямоугольник, треугольник, окружность. Клиент имеет возможность просмотра информации, редактирования, удаления, получения информации о площади и периметре указанных фигур. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

8. Сервер хранит два одномерных массива строк фиксированной длины. Клиент имеет возможность обращения к отдельным строкам массивов по индексам, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывода на экран элемента массива по заданному индексу и всех массивов, редактирования массивов. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

9. Сервер хранит предметный указатель, каждая компонента которого содержит слово и номера страниц, где это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Клиент имеет возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

10. Сервер хранит некоторую матрицу произвольного размера. Клиент имеет возможность формирования элементов матрицы, редактирования элементов матрицы, изменения числа строк и столбцов матрицы, вывода на экран подматрицы любого размера и всей матрицы. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

11. Сервер хранит информацию о клиентах некоторой торговой организации. Клиент имеет возможность добавления, удаления, редактирования, сортировки информации на сервере, а также поиска по заданным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

12. Сервер хранит одномерные массивы целых чисел (вектора). Клиент имеет возможность формирования элементов векторов, редактирования, удаления, сортировки, а также поэлементного сложения и вычитания векторов с одинаковыми границами индексов, умножения и деления всех элементов векторов на скаляр, вывода максимального и минимального элементов векторов. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

13. Север хранит информацию о результатах студенческой сессии. Клиент имеет возможность просмотра, редактирования, удаления информации, поиска студента по какому-либо признаку (например, по фамилии, оценкам по предмету), сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

14. Сервер хранит информацию о наличии билетов к кассе кинотеатра. Клиент имеет возможность просмотра, добавления, удаления информации, сортировки по разным полям, а также поиска по заданному условию (например, по фильму, по стоимости билета, по времени начала сеанса). Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

15. Сервер хранит список вопросов по некоторой дисциплине (например, компьютерные сети), и варианты ответов на них (на каждый вопрос – три ответа, среди которых только один правильный). Клиент имеет возможность просмотра вопросов и выбора правильного ответа на них. Сервер проверяет правильность ответов клиента и выставляет ему оценку. Клиент также имеет возможность редактирования вопросов и ответов на сервере. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

4.7 Контрольные вопросы

1. Какие преимущества предоставляет параллельная реализация сервера?
2. Какие основные задачи выполняет ведущий процесс?
3. Для чего используется блокирующий вызов функции *accept()*?
4. Когда и для чего вызывается функция *fork()*?
5. Каким образом решается проблема не полностью завершившихся процессов (процессов, информация о которых остается в системных таблицах)?
6. Какой код используется в процессе для обозначения нормального завершения?

ЛАБОРАТОРНАЯ РАБОТА №5

Создание однопотоковых параллельных серверов TCP

Цель работы: изучить методы создания приложений, используя алгоритм обработки запросов однопотоковыми параллельными серверами.

В работе рассматриваются однопотоковые параллельные серверы, в основе которых лежит использование асинхронного ввода/вывода, организованного с помощью функции *select* операционной системы.

5.1 Однопотоковые параллельные серверы

В приложениях клиент/сервер, характеризующихся тем, что затраты на обеспечение ввода/вывода превышают затраты на подготовку ответа на запрос, в сервере может использоваться асинхронный ввод/вывод для организации псевдопараллельной работы клиентов. В основе этого подхода лежит простой принцип: необходимо предусмотреть, чтобы единственный поток выполнения в сервере держал открытыми соединения *TCP* с несколькими клиентами и обеспечивал обслуживание сервером того соединения, через которое в определенный момент поступают данные. Таким образом, сам факт поступления данных используется для активизации обработки данных сервером. С теоретической точки зрения в основе организации работы сервера лежит использование механизма квантования времени операционной системы для разделения ресурсов процессора между потоками и, следовательно, между соединениями.

Однако на практике в некоторых серверах (например службы *ECHO*) квантование времени применяется редко. В этих случаях для активизации обработки используется сам факт поступления данных. В основе такого решения лежит изучение структуры потоков данных, проходящих по объединенной сети. Данные поступают на сервер в виде пульсирующего трафика, а не устойчивого потока, поскольку базовая объединенная сеть доставляет данные в отдельных пакетах. Неравномерность трафика еще больше возрастает в связи с тем, что клиенты, как правило, отправляют данные в виде крупных блоков, чтобы каждый из сформированных в результате сегментов *TCP* мог занять всю дейтаграмму *IP*. На сервере каждый ведомый поток основную часть времени проводит в состоянии, заблокированном в вызове функции *read* (или *recv*), ожидая поступления следующей порции данных. Сразу после поступления данных происходит возврат управления из функции *read* (*recv*), и выполнение ведомого потока возобновляется. Ведомый поток вызывает функцию *write* (*send*) для передачи данных обратно клиенту, а затем снова вызывает функцию *read* (*recv*) для перехода в состояние ожидания поступления следующей порции данных. Процессор сможет выдержать нагрузку по обработке данных, поступающих от многочисленных клиентов, не внося замедления, только в том случае, если он

будет способен завершить цикл чтения и записи одного потока до поступления данных для другого ведомого потока.

Безусловно, если нагрузка становится столь значительной, что процессор не может завершить выполнение одного запроса до поступления другого, принцип организации работы с квантованием времени становится более приемлемым. Операционная система переключает процессор между всеми ведомыми потоками, имеющими данные, предназначенные для обработки. Однако в случае простых служб, требующих небольшого объема обработки данных для обслуживания каждого запроса, велика вероятность того, что для активизации процесса обработки будет использоваться сам факт поступления данных.

Параллельные серверы, в которых на обработку каждого запроса требуется мало времени, часто выполняют обработку данных последовательно; при этом причиной активизации процесса обработки является сам факт поступления данных. Квантование времени начинает использоваться только с того момента, когда нагрузка становится настолько высокой, что процессор не может обрабатывать запросы последовательно.

Поскольку в результате изучения характера работы параллельного сервера стало очевидно, что он часто выполняет обработку данных последовательно, был сделан вывод, что для выполнения той же задачи может применяться один поток. Допустим, что один поток сервера обслуживает соединения *TCP* одновременно с несколькими клиентами. Поток блокируется, ожидая поступления данных. Сразу после поступления данных через любое соединение поток активизируется, обрабатывает запрос и передает ответ. Затем он снова блокируется, ожидая поступления данных из другого соединения. При условии, что процессор работает достаточно быстро для того, чтобы выдержать нагрузку, возложенную на сервер, однопотоковая версия сможет обслуживать соединения с таким же успехом, как и версия с несколькими потоками. Действительно, однопотоковая реализация не требует переключения между контекстами потоков или процессов, поэтому она может даже выдержать более высокую нагрузку по сравнению с реализацией, в которой используются несколько потоков или процессов.

5.2 Использование функции select

В основе разработки программы однопотокового, параллельного сервера лежит использование асинхронного ввода/вывода, организованного с помощью функции *select* операционной системы. Сервер создает сокет для каждого соединения, которое он должен поддерживать, а затем вызывает функцию *select*, которая ожидает поступления данных через каждое из них. По существу, функция *select* может ожидать поступления запросов на выполнение операций ввода/вывода через все возможные сокеты, поэтому она может также одновременно ожидать поступления новых запросов на установление соединения.

На рисунке 6 показана схема организации работы и состав сокетов однопотокового, параллельного сервера. Для управления обмена данными через все сокеты применяется один поток.

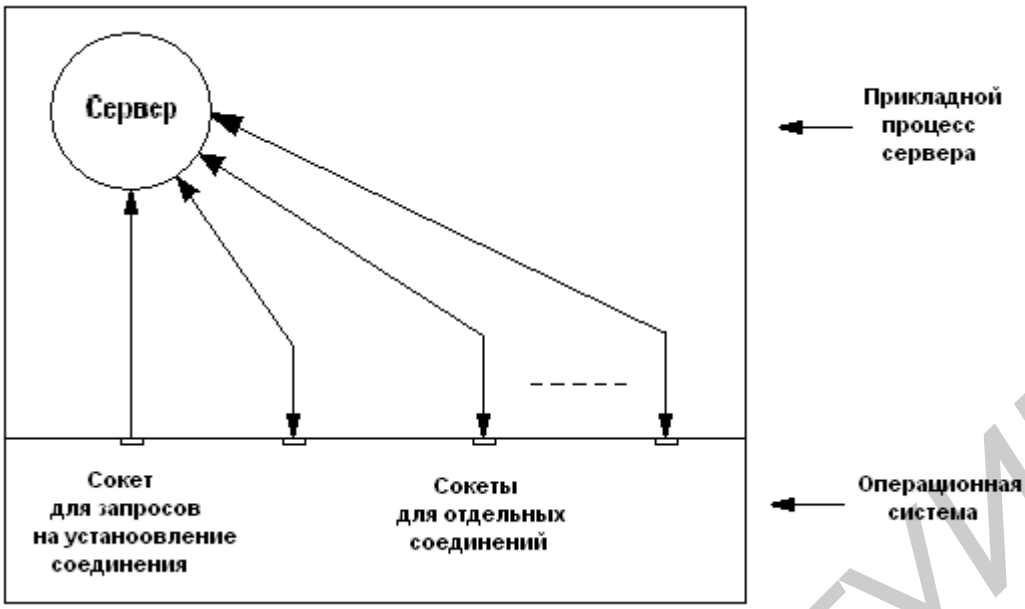

Рисунок 6 – Схема организации процессов сервера с установлением логического соединения, в котором параллельная работа обеспечивается с использованием одного потока, управляющего сразу несколькими сокетами

По сути, один единственный поток должен выполнять обязанности и ведущего, и всех ведомых потоков. Он сопровождает набор сокетов, причём один из них (так называемый ведущий сокет) остается привязанным к общепринятому порту, через который он должен принимать запросы на установление соединения. Каждый из ведомых сокетов в наборе соответствует соединению, для которого ведомый поток должен обрабатывать запросы. Сервер передает набор дескрипторов сокетов функции *select* в качестве параметра и ожидает активизации любого из них. После возврата управления функция *select* передает в вызывающий оператор битовую маску, которая указывает, какой из дескрипторов в наборе готов к работе. В сервере для принятия решения о том, с каким из дескрипторов нужно продолжить работу, используется порядок их активизации.

Для определения того, какие операции (ведущего или ведомого потока) должны быть выполнены для данного дескриптора, в однопотоковом сервере используется сам дескриптор. Если к работе готов дескриптор, соответствующий ведущему сокету, сервер выполняет для него такие же операции, какие выполнил бы ведущий поток: он вызывает функцию *accept* с этим сокетом для получения нового соединения. Если же к работе готов дескриптор, соответствующий ведомому сокету, сервер выполняет операцию ведомого потока: он вызывает функцию *read (recv)* для получения запроса, а затем отвечает на этот запрос.

5.3 Методические указания по созданию однопотокового параллельного сервера с установлением логического соединения

Рассмотрим процесс создания такого сервера на следующем примере.

Осуществить взаимодействие клиента и сервера на основе протокола TCP. Реализовать параллельное соединение с использованием одного потока. На сервере хранится список студентов. Каждая запись списка содержит следующую информацию о студенте:

- ФИО студента;
- номер группы;
- размер стипендии;
- оценки по N предметам.

Таких записей должно быть не менее 5-ти.

По личному запросу клиент получает от сервера список только тех студентов, которые не имеют оценки 3. Предусмотреть возможность редактирования записей списка клиентом.

Рассмотрим серверную часть.

Серверная часть

```
#include <sys/types.h>
#include<sys/socket.h>
#include<sys/signal.h>

//#include<sys/resource.h>
#include<netinet/in.h>

#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <netdb.h>

const int
          NameL = 33,
          GroupL = 10,
          payL =10;

struct Student{
     char name[NameL+1];
     char group[GroupL+1];
     char pay[payL+1];
     char mark[5];
} st[5];

int DoProcess(int fd){
puts("rabotaet doProcess");
     long int i,num,t,newS;
```

```
          int m;
          char p;
          char buf[256],b[256];
          while (true){
               recv(newS,buf,sizeof(buf),0);

               p=buf[0];
               puts(buf);
               switch(p){
               case '1':
                    buf[0]='\0';
                    for (i=1;i<=5;i++)
                    if (st[i].mark[1]!='3' && st[i].mark[2]!='3' &&
st[i].mark[3]!='3' z&& st[i].mark[4]!='3'){
                              strcat(buf,st[i].name);
                              strcat(buf,"\n");
                         }
                    send(newS,buf,sizeof(buf),0);
                    puts(buf);
                    break;
               case '2':
                    recv(newS,buf,sizeof(buf),0);
                    num=atoi(buf);
                    printf("%d\n",num);
                    recv(newS,buf,sizeof(buf),0);
                    strcpy(st[num].name,buf);

                    recv(newS,buf,sizeof(buf),0);
                    strcpy(st[num].group,buf);

                    recv(newS,buf,sizeof(buf),0);
                    strcpy(st[num].pay,buf);

                    recv(newS,buf,sizeof(buf),0);
                    st[num].mark[1]=buf[0];
                    recv(newS,buf,sizeof(buf),0);
                    st[num].mark[2]=buf[0];
                    recv(newS,buf,sizeof(buf),0);
                    st[num].mark[3]=buf[0];
                    recv(newS,buf,sizeof(buf),0);
                    st[num].mark[4]=buf[0];

                    break;
               case '3':
                    for (i=1;i<=5;i++){
                       buf[0]='\0';
                       strcat(buf,st[i].name);strcat(buf," ");
                       strcat(buf,st[i].group);strcat(buf," ");
                       strcat(buf,st[i].pay);strcat(buf," ");
                       b[0]=st[i].mark[1];
                       strcat(buf,b);strcat(buf," ");
                       b[0]=st[i].mark[2];
```

```
                strcat(buf,b);strcat(buf," ");
                b[0]=st[i].mark[3];
                strcat(buf,b);strcat(buf," ");
                b[0]=st[i].mark[4];
                strcat(buf,b);strcat(buf,"\n");

                send(newS,buf,sizeof(buf),0);
            }
            break;
            case '4':
                 exit(0);
            }
        }
}

int main(){
            strcpy(st[1].name,"Ivanov Ivan Ivanovich");
            strcpy(st[1].group,"111111");
            strcpy(st[1].pay,"111111");
            st[1].mark[1]='4';
            st[1].mark[2]='2';
            st[1].mark[3]='5';
            st[1].mark[4]='5';

            strcpy(st[2].name,"petrov petr petrovich");
            strcpy(st[2].group,"222222");
            strcpy(st[2].pay,"222222");
            st[2].mark[1]='4';
            st[2].mark[2]='2';
            st[2].mark[3]='4';
            st[2].mark[4]='5';

            strcpy(st[3].name,"sidorov sidor sidorovich");
            strcpy(st[3].group,"333333");
            strcpy(st[3].pay,"333333");
            st[3].mark[1]='5';
            st[3].mark[2]='1';
            st[3].mark[3]='4';
            st[3].mark[4]='2';

            strcpy(st[4].name,"brusnikin igor igorevich");
            strcpy(st[4].group,"444444");
            strcpy(st[4].pay,"444444");
            st[4].mark[1]='2';
            st[4].mark[2]='3';
            st[4].mark[3]='5';
            st[4].mark[4]='2';

            strcpy(st[5].name,"klukvin yurij yurjevich");
            strcpy(st[5].group,"555555");
            strcpy(st[5].pay,"555555");
            st[5].mark[1]='4';
```

```
        st[5].mark[2]='1';
        st[5].mark[3]='1';
        st[5].mark[4]='3';

        fd_set rfds;
        fd_set afds;

        struct sockaddr_in sin;
        int s;

        memset(&sin,0,sizeof(sin));
        sin.sin_family=AF_INET;
        sin.sin_port=htons(7500);
        sin.sin_addr.s_addr=htonl(INADDR_ANY);


        int alen;
        int fd,nfds;

        s=socket(AF_INET, SOCK_STREAM,0);
        bind(s,(struct sockaddr *)&sin, sizeof(sin));
        listen(s,5);

        nfds=getdtablesize();
        FD_ZERO(&afds);
        FD_SET(s,&afds);
        int newS;
        int k;
        while(true){
             memcpy(&rfds,&afds,sizeof(rfds));
        select(nfds,&rfds,NULL,NULL,(struct timeval*)0);
             if (FD_ISSET(s,&rfds)){
             alen=sizeof(sin);
             int temp=(int)alen;
             newS=accept(s,(struct sockaddr*)&sin,&temp);
             FD_SET(s,&afds);
             }

             for (fd=0;fd<nfds;fd++)
                  if (fd!=s && FD_ISSET(fd,&rfds))
                       if (DoProcess(fd)==0){
                            (void)close(fd);
                            FD_CLR(fd,&afds);
                       }
        }
        return 0;
}
```

5.4 Индивидуальные задания

Разработать приложение, реализующее архитектуру «клиент-сервер». Необходимо реализовать параллельный сервер с установлением логического соединения (*TCP*), использующий отдельный процесс для обработки запросов клиентов. Логику взаимодействия клиента и сервера реализовать так, как указано в индивидуальном задании.

1. Сервер хранит информацию о домашней библиотеке. Клиент имеет возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

2. Сервер хранит информацию из записной книжки. Клиент имеет возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

3. Сервер хранит информацию о студенческой группе. Клиент имеет возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

4. Сервер хранит информацию о некоторой коллекции компакт-дисков. Клиент имеет возможность просмотра, добавления, удаления информации, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

5. Сервер хранит информацию о лицевых счетах вкладчиков банковского учреждения. Запись о каждом вкладчике состоит из информации: фамилия, дата операции, сумма вклада, общая сумма. Клиент имеет возможность пополнения и изъятия данных о вкладчиках, ведения текущих счетов, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

6. Сервер хранит информацию о персонале: фамилия, должность, зарплата, дата рождения. Клиент имеет возможность добавления, изменения, удаления информации, сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

7. Сервер хранит информацию о плоских геометрических фигурах: прямоугольник, треугольник, окружность. Клиент имеет возможность просмотра информации, редактирования, удаления, получения информации о площади и периметре указанных фигур. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

8. Сервер хранит два одномерных массива строк фиксированной длины. Клиент имеет возможность обращения к отдельным строкам массивов по индексам, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывода на экран элемента массива по заданному индексу и всех массивов, редактирования массивов. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

9. Сервер хранит предметный указатель, каждая компонента которого содержит слово и номера страниц, где это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Клиент имеет возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

10. Сервер хранит некоторую матрицу произвольного размера. Клиент имеет возможность формирования элементов матрицы, редактирования элементов матрицы, изменения числа строк и столбцов матрицы, вывода на экран подматрицы любого размера и всей матрицы. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

11. Сервер хранит информацию о клиентах некоторой торговой организации. Клиент имеет возможность добавления, удаления, редактирования, сортировки информации на сервере, а также поиска по заданным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

12. Сервер хранит одномерные массивы целых чисел (вектора). Клиент имеет возможность формирования элементов векторов, редактирования, удаления, сортировки, а также поэлементного сложения и вычитания векторов с одинаковыми границами индексов, умножения и деления всех элементов векторов на скаляр, вывода максимального и минимального элементов векторов. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

13. Север хранит информацию о результатах студенческой сессии. Клиент имеет возможность просмотра, редактирования, удаления информации, поиска студента по какому-либо признаку (например, по фамилии, оценкам по предмету), сортировки по разным полям. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

14. Сервер хранит информацию о наличии билетов к кассе кинотеатра. Клиент имеет возможность просмотра, добавления, удаления информации, сортировки по разным полям, а также поиска по заданному условию (например, по фильму, по стоимости билета, по времени начала сеанса). Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

15. Сервер хранит список вопросов по некоторой дисциплине (например компьютерные сети), и варианты ответов на них (на каждый вопрос – три отве-

та, среди которых только один правильный). Клиент имеет возможность просмотра вопросов и выбора правильного ответа на них. Сервер проверяет правильность ответов клиента и выставляет ему оценку. Клиент также имеет возможность редактирования вопросов и ответов на сервере. Программа клиента должна содержать меню, позволяющее осуществлять указанные действия на сервере.

5.5 Контрольные вопросы

1. Принцип работы механизма квантования времени организации работы сервера.
2. В каком виде данные поступают на сервер в том случае, если метод квантования времени не применяется?
3. Что происходит после поступления на сервер всех данных?
4. Для чего используется функция *write* (*send*)?
5. При каком условии процессор сможет выдержать нагрузку по обработке данных, поступающих от многочисленных клиентов, не внося никаких замедлений?
6. В каком случае используется функция *select*?

Лабораторная работа №6

Создание параллельного сервера с установлением логического соединения с помощью пула потоков/процессов

Цель работы: изучить методы создания серверов, используя алгоритм параллельной обработки запросов с пулом готовых потоков/процессов.

В работе приведен пример параллельного сервера с установлением логического соединения. Для обеспечения параллельного формирования ответов на запросы клиентов сервер использует несколько (пул) заранее подготовленных потоков/процессов.

6.1 Пул потоков/процессов

В тех случаях, когда затраты времени на создание процесса являются значительными, может применяться простой метод сокращения времени, ограничения максимальной степени распараллеливания и обеспечения высокой производительности параллельных серверов. В основе этого метода лежит предварительное создание параллельных процессов или потоков, что позволяет исключить издержки в процессе работы их создания. После создания сервера потоки продолжают функционировать.

Что такое пул потоков? В жизни мы очень часто встречаемся с организацией пула. Например, когда вы идете в столовую, вы встречаетесь с пулом подносов. Клиентов может быть намного меньше, чем подносов, и наоборот. Когда подносов много, они лежат без дела; когда подносов мало, клиенты ждут, пока они освободятся. Число подносов, то есть размер пула, заранее определяется так, чтобы в большинстве случаев клиенты не ждали подносов. Однако случаются часы пик, когда клиентов очень много. Просто нереально выделить отдельный поднос каждому клиенту, да и не нужно это. Клиент все равно будет стоять в очереди к кассе, так что траты на подносы не принесут реальных выгод. Это, конечно, очень далекая и несовершенная аналогия, но она показывает, что в природе и жизни пул чего-либо очень часто используется как наиэффективнейшая схема обслуживания запросов.

При использовании метода предварительного создания разработчик предусматривает создание в программе ведущего сервера N ведомых процессов или потоков с началом ее выполнения (рисунок 7). Каждый ведомый процесс использует средства, доступные в операционной системе, для перехода в режим ожидания поступления запроса. После появления запроса один из ожидающих ведомых процессов начинает выполнение и обрабатывает этот запрос. После завершения обработки запроса ведомый процесс не заканчивает свою работу, а возвращается к выполнению кода, предусматривающего ожидание очередного запроса.

Основное преимущество предварительного создания обусловлено снижением издержек операционной системы. Поскольку серверу не приходится

ждать создания ведомого процесса после поступления запроса, он обрабатывает запросы быстрее. Этот метод приобретает особое значение, если на обработку запроса требуется длительное время на выполнение более продолжительных операций ввода-вывода.

Однако такой метод имеет и недостатки: необходимо соблюдать крайнюю осторожность при использовании ресурсов. Чтобы понять, с чем это связано, рассмотрим постоянный ведомый процесс, в котором предусмотрено распределение небольшого объема памяти при поступлении каждого запроса, но после завершения обработки запроса освобождение памяти не происходит из-за ошибки в программе. Хотя такая проблема может появиться не сразу, со временем данный ведомый процесс может исчерпать все адресное пространство.

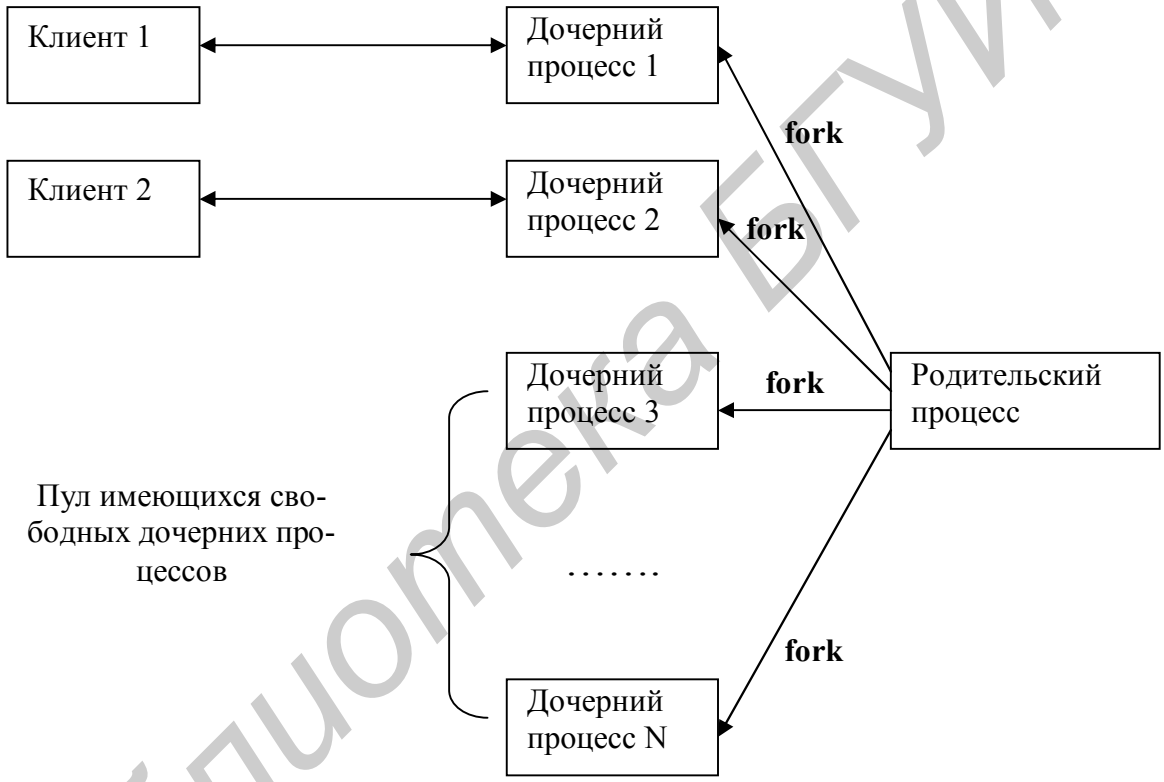

Рисунок 7 – Предварительное создание сервером дочерних процессов

6.2 Методические указания по созданию параллельного сервера с установлением логического соединения TCP, использующего пул потоков для обработки запросов клиента

Теперь приведем текст примера сервера *TCP* – параллельного сервера с установлением логического соединения с пулом потоков. Данный пример реализует прикладной протокол *DAYTIME*.

Новый поток может быть создан в любое время путем вызова функции *pthread_create*. Операционная система ограничивает максимально допустимое

количество параллельных потоков, равно как и максимально допустимое количество параллельных процессов.

Серверная часть

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

char * daytime() {
    time_t now;
    now=time(NULL);
    return ctime(&now);
}

void * udp_thread(int * p_sock) {
    int sock=*p_sock;
    for (;;) {
     struct sockaddr from;
     int len=sizeof(from);
     char buf[81];
     memset(buf,0,81);
     recvfrom(sock,buf,80,0,&from,&len);
     printf("udp incoming:%s",buf);

     memset(buf,0,81);
     strncpy(buf,daytime(),80);

     sendto(sock,buf,strlen(buf),0,&from,len);
     puts("answer udp");
    }

    return NULL;
}

void * tcp_thread(int *p_sock) {
    int sock=*p_sock;
    for (;;) {
     int c_sock=accept(sock,NULL,NULL);
          char buf[81];
     memset(buf,0,81);
     strncpy(buf,daytime(),80);
     write(c_sock,buf,strlen(buf));
     shutdown(c_sock,0);
     close(c_sock);
```

```
     puts("answer tcp");
    }
    return NULL;
}

int main() {
    struct sockaddr_in addr;

    memset(&addr,0,sizeof(addr));

    addr.sin_family=AF_INET;
    addr.sin_port=htons(1212);
    addr.sin_addr.s_addr=INADDR_ANY;

    int sock;
    const int threads=10;
    pthread_t tid;

    if ( fork() ) {
     sock=socket(PF_INET,SOCK_STREAM,0);
     bind(sock,(struct sockaddr *)&addr,sizeof(addr));
     listen(sock,5);
     for (int i=0; i<threads; i++)/*создаем 10 потоков*/
         pthread_create(&tid,NULL,&tcp_thread,&sock);
    }
    else {
     sock=socket(PF_INET,SOCK_DGRAM,0);
     bind(sock,(struct sockaddr *)&addr,sizeof(addr));
     for (int i=0; i<threads; i++)/*создаем 10 потоков*/
         pthread_create(&tid,NULL,&udp_thread,&sock);
    }

    pthread_join(tid,NULL); //возобновление работы потока

    return 0;
}
```

6.3 Индивидуальные задания

Разработать приложение, реализующее архитектуру «клиент-сервер». Необходимо реализовать параллельный сервер с установлением логического соединения (TCP), использующего пул потоков для обработки запросов клиентов. Логику взаимодействия клиента и сервера реализовать следующим образом.

1. Разработать приложение-генератор случайных чисел. На клиентской части вводится целое положительное число N и передается серверу, а тот в свою очередь возвращает клиенту массив случайных чисел от 1 до N.

2. Разработать приложение-поисковик слов. На сервере хранится определенный текст. На клиентской части вводится слово для поиска и передается серверу, а тот в свою очередь осуществляет поиск этого слова в тексте и возвращает клиенту все предложения, в которых встречается это слово.

3. Разработать приложение-счетчик букв. На клиентской части вводится строка и передается серверу, а тот в свою очередь осуществляет подсчет гласных и согласных букв и возвращает этот результат клиенту.

4. Разработать приложение-определитель матрицы. На клиентской части вводится исходная матрица произвольного порядка и передается серверу, а тот в свою очередь вычисляет определитель этой матрицы и возвращает результат клиенту.

5. Разработать приложение для нахождения обратной матрицы размером 3x3. Исходная матрица вводится на клиентской части и передается серверу, а тот в свою очередь возвращает клиенту обратную матрицу.

6. Разработать приложение для определения счастливчика лотереи. На сервере хранятся номера билетов. На каждом билете имеются 10 случайных чисел от 1 до 100. На клиентской части вводятся 10 чисел от 1 до 100, и сервер должен определить номер билета, в котором имеется больше всего совпадений с введенными числами.

7. Разработать приложение для определения призовых мест на соревнования по прыжкам в длину. На сервере хранятся фамилии участников соревнований, их идентификационные номера. На клиентской части вводятся результаты прыжков по каждому идентификационному номеру, а сервер возвращает фамилии спортсменов, занявших 1, 2 и 3 места.

8. Разработать приложение по поиску квартиры для покупки. Стоимости квартир и их адреса хранятся на сервере. На клиентской части вводится предельная сумма для покупки квартиры, а сервер возвращает клиенту адреса всех квартир с такой или меньшей стоимостью.

9. Разработать программу учета продаж журналов и газет. Вся информация о журналах и газетах находится на сервере. Клиент делает запрос на сервер с указанием действий, которые он хочет осуществить. Клиент может осуществить следующие действия: заказ на покупку журналов или газет, просмотр непроданных журналов и газет, возможность редактирования информации о журналах и газетах.

10. Разработать программу учета проданных товаров продуктового магазина. На сервере хранится информация о наличии некоторых товаров. Клиент вводит с клавиатуры запрос на покупку того или иного товара. Назад он получает результат (информацию о том, что товар отмечен как купленный). Клиент имеет возможность просматривать на сервере информацию о товарах (наименование товаров, исходное количество, проданное количество, остаток на складе), также может ее редактировать, добавлять и удалять.

11. Разработать программу учета сдачи экзаменов студентами. На сервере хранится информация о номерах групп студентов, ФИО студентов, сдаваемых студентами предметах. Клиент имеет возможность заполнения ведомостей студентов по тому или иному предмету, редактирования ведомостей, их просмотра.

12. Разработать программу поиска фрагмента текста. На сервере хранится исходный текст. Клиент имеет возможность ввести у себя некоторый искомый

фрагмент текста и получить от сервера результат поиска (есть или нет такой фрагмент в тексте сервера). Клиент имеет возможность редактирования исходного текста на сервере.

13. Разработать программу учета проданных мобильных телефонов. На сервере хранится информация о наличии некоторых мобильных телефонов в магазине. Клиент вводит с клавиатуры запрос на покупку того или иного мобильного телефона. Назад он получает результат (информацию о том, что товар отмечен как купленный). Клиент имеет возможность просматривать на сервере информацию о телефонах (наименование телефона, его модель, исходное количество, проданное количество, остаток на кладе), также может ее редактировать, добавлять и удалять.

14. Разработать программу учета проданных билетов на железной дороге. Клиент посылает на сервер запрос на просмотр проданных, забронированных или свободных билетов. В ответ от сервера он получает необходимую информацию. Клиент имеет возможность на сервере редактировать информацию о билетах (номера поезда, время отправки, цена билетов).

15. Разработать программу контроля качества знаний у студентов при выполнении теста. Клиент посылает на сервер некоторый выполненный им тест (условно можно взять 5 вопросов с ответами на них). Программа сервера проверяет результат и посылает оценку клиенту. При количестве ошибок менее 2-х ставится оценка 10, более 1-й и менее 4-х ошибок – оценка 7, более 3 ошибок - оценка 4. Определить минимальное и максимальное количество ошибок.

6.4 Контрольные вопросы

1. Что лежит в основе метода распараллеливания потоков/процессов?
2. Каковы принципы работы пула потоков?
3. В чем заключаются преимущества данного метода?
4. Главные недостатки данного метода?
5. С помощью каких функций производится работа с пулом потоков?

ЛИТЕРАТУРА

1 Пынькин, Д. А. Системное программное обеспечение локальных вычислительных сетей : учеб.-метод. комплекс / Д. А. Пынькин, И. И. Глецевич [электронный ресурс]. – Минск : 2006. http://abitur.bsuir.by/m/12_116608_1_49901.pdf.

2 http://cpprog.narod.ru/articles.html

3 Олифер, В. Г. Компьютерные сети. Принципы, технологии, протоколы / В. Г. Олифер, Н. А. Олифер. – СПб., Питер, 2002**.**

4 Таненбаум, Э. Компьютерные сети / Э. Таненбаум. – СПб.: Питер, 2002.

5 Дилип, Н. Стандарты и протоколы Интернета / Н. Дилип; пер. с англ. – М. : Издательский отдел «Русская Редакция» ТОО «Channel Trading Ltd.», 1999. – 672 с.

6 Щербо, Б. М. Стандарты по локальным вычислительным сетям : справочник / Б. М. Щербо, В. К. Киреичев, С. И. Самойленко ; под ред. С. И. Самойленко. – М. : Радио и связь, 1990.

7 Снейдер, Й. Эффективное программирование TCP/IP. Библиотека программиста / Й. Снейдер. – СПб.: Питер, 2002.

8 Компьютерные сети. Учебный курс. – М. : «Русская Редакция» ТОО «Channel Trading». 1997.

9 Челлис, Дж. Основы построения сетей : учеб. пособие для специалистов MCSE / Дж. Челлис, Ч. Перкинс, М. Стриб. – М. : Лори 1997.

10 Петерсен, Р. LINUX : руководство по операционной системе: В 2 т. / Р. Петерсон ; пер. с англ. – 2-е изд., перераб. и доп. – Киев : Издательская группа BHV, 1999.

11 Кью, П. Использование UNIX. Специальное издание / П. Кью ; пер. с англ. – М. : СПб., Киев : Издательский дом «Вильямс», 1999.

12 Сетевые средства Microsoft Windows NT Server 4.0 ; пер. с англ. – СПб. : BHV – Санкт-Петербург, 1998.

13 Ларионов, А. М. Вычислительные комплексы, системы и сети : учебник для вузов / С. А. Майоров, Г. И. Новиков – Л. : Энергоатомиздат, 1987.

14 Шпаковский, Г. И. Архитектура параллельных ЭВМ : учеб. пособие для вузов / Г. И. Шпаковский. – Минск : Университетское, 1989.

15 Кулаков, Ю. А. Компьютерные сети / Ю. Л. Кулаков, Г. М. Луцкий. – Киев : ЮНИОР, 1998.

Св. план 2011, поз. 132

Учебное издание

Комличенко Виталий Николаевич
Федосенко Владимир Алексеевич
Унучек Татьяна Михайловна и др.

КОМПЬЮТЕРНЫЕ СЕТИ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *Т. П. Андрейченко*
Корректор *Е. Н. Батурчик*

Подписано в печать 29.05.2013. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 4,53. Уч.-изд. л. 4,0. Тираж 150 экз. Заказ 451.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6