ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего профессионального образования

Санкт-Петербургский государственный университет аэрокосмического приборостроения

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторной работы 4

по курсу «Операционные системы»

"Сетевое взаимодействие"

Прокофьева Т.Л.

Санкт-Петербург

2022

**ЦЕЛЬ РАБОТЫ**

1. Изучить понятие сокетов.
2. Изучить механизм межсетевого взаимодействия на основе сокетов.
3. Научиться разрабатывать сетевые приложения типа клиент-сервер с использованием протоколов UDP и TCP.

**ПОНЯТИЕ СОКЕТОВ КАК СРЕДСТВА МЕЖПРОЦЕССОРНОГО И МЕЖСЕТЕВОГО ВЗАИМОДЕЙСТВИЯ**

Какие бы замечательные идеи в области телекоммуникаций, распределенных баз знаний или поисковых систем вам не пришли в голову, реализовать их на практике можно, лишь написав соответствующую программу. Основные операционные среды (\*nix и Windows 2000+) содержат базовый механизм межсетевого взаимодействия на идеологии соединителей (socket). Эта технология была разработана в университете г. Беркли (США) для системы Unix, поэтому соединители иногда называют соединителями Беркли (Berkeley Sockets). Соединители реализуют механизм взаимодействия не только партнеров по телекоммуникациям, но и процессов в ЭВМ вообще.

Socket API был впервые реализован в операционной системе Berkeley UNIX. Сейчас этот программный интерфейс доступен практически в любой модификации Unix, в том числе в Linux. Хотя все реализации чем-то отличаются друг от друга, основной набор функций в них совпадает. Изначально интерфейс сокетов был доступен, в основном, для программ на языках C, C++ и т.п., но в настоящее время средства для работы с ними предоставляют многие языки (Perl, PHP, Python, Java и др.).

Сокеты предоставляют весьма мощный и гибкий механизм межпроцессного взаимодействия (IPC – Inter-Process communication). Они могут использоваться для организации взаимодействия программ (в дальнейшем *процессов*) в рамках локального компьютера, локальной сети или посредством Internet. Это позволяет вам создавать распределённые приложения различной сложности. Кроме того, одним из самых главных достоинств данного интерфейса заключается в том, что он кроссплатформенный и позволяет организовывать взаимодействие между процессами, запущенными на различных ОС. Базовый интерфейс Berkeley Sockects позволяет без особых затруднений портировать программы с одной ОС на другую.

В общем случае сокеты поддерживают многие стандартные сетевые протоколы (конкретный их список зависит от реализации) и предоставляют унифицированный интерфейс для работы с ними. Но наиболее широко распространены сокеты для работы в IP-сетях.

Как видим, сокеты – весьма мощное и удобное средство для сетевого программирования. Понимание принципа работы с сокетами позволяет легко реализовывать всевозможные сетевые приложения, поддерживающие как стандартные протоколы, так и специализированные.

**ИСПОЛЬЗОВАНИЕ СОКЕТОВ**

Сокет (Socket) – это конечная точка сетевых коммуникаций. Он является чем-то вроде «портала», через которое можно отправлять байты во внешний мир. Приложение просто пишет поток данных в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.

В программе сокет идентифицируется дескриптором. Программа получает дескриптор от операционной системы при создании сокета, а затем передаёт его сервисам Socket API для указания сокета, над которым необходимо выполнить то или иное действие.

# Программирование сокетов: базовые сведения

В классическом варианте ядро сетевого приложения состоит из 2-х программ: клиента и сервера. Когда эти программы запускаются, создаются клиентский и серверный процессы, которые взаимодействуют друг с другом, обмениваясь сообщениями через сокеты.

Клиент-сервер (англ. Сlient/Server) – сетевая архитектура, в которой устройства являются либо клиентами, либо серверами. Клиентом (front end) является запрашивающая машина (обычно ПК), сервером (back end) – машина, которая отвечает на запрос. Оба термина (клиент и сервер) могут применяться как к физическим устройствам, так и к программному обеспечению.

Существует два вида приложений с архитектурой клиент/сервер. К первому виду относятся приложения, поддерживающие стандартный протокол, описанный в документах RFC. В этом случае как серверная, так и клиентская части приложения должны соответствовать всем описаниями, приведенным в RFC. Например, если приложение использует протокол FTP (его будем рассматривать на следующей лабораторной работе), то клиентская и серверная программы должны быть построены в соответствии с требованиями RFC 959. Следование стандартам позволяет независимым разработчикам создавать различные части одного и того же приложения. Например, web-браузер Mozilla способен успешно взаимодействовать с web-сервером IIS, а встроенный Internet Explorer ftp-клиент успешно взаимодействует с FTP-сервером, работающем на платформе ОС Linux и т.д.

Второй вид приложений с архитектурой клиент/сервер – нестандартные приложения. В таких приложениях поддержка клиентом и сервером каких-либо согласованных с RFC стандартов не требуется. При этом для разработки клиента и сервера, способного работать друг с другом разработчикам в любом случае требуется знать (или разработать) протокол взаимодействия.

Наиболее распространенными и удобными для применения транспортными протоколами для прикладных сетевых приложений являются уже упоминавшиеся на предыдущих лабораторных работах протоколы TCP и UDP.

Выбор того или иного протокола зависит напрямую от решаемой задачи. Основные области применения протоколов таковы: TCP используется для передачи потоковых данных, где достоверность получения корректных данных имеет высокое значение; UDP применим при организации взаимодействия посредством отдельных, зачастую не связанных друг с другом датаграмм.

На данной лабораторной работе мы познакомимся с принципом создания клиент/серверных приложений с использованием TCPсокетов.

# Программирование TCP-сокетов

Как уже отмечалось ранее, сетевые процессы взаимодействуют друг с другом при помощи сообщений, посылаемых через сокеты.

Рассмотрим процесс взаимодействия клиента и сервера более подробно. В функции клиента входит инициирование соединения с сервером, а сервер должен быть готовым к установлению соединения. Это означает, что, во-первых, программа-сервер должна быть запущена раньше, чем клиент сделает попытку установить соединения, и, во-вторых, что сервер должен располагать сокетом, с помощью которого устанавливается соединение.

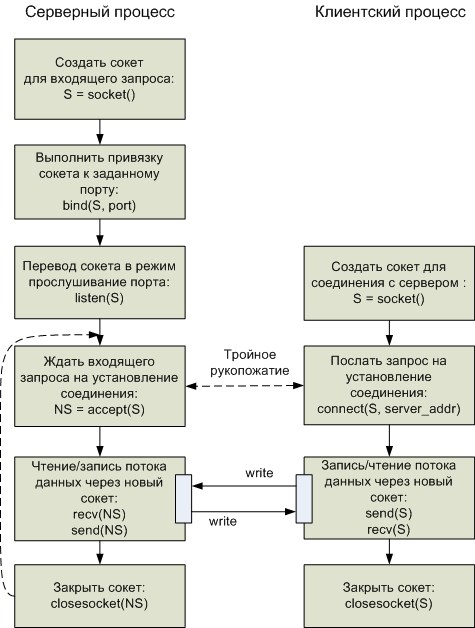
Первым действие клиентской программы является создание сокета, при этом программа указывает адрес серверного процесса, состоящий из IP-адреса, протокола (TCP) и порта процесса. После создания сокета клиентская сторона протокола TCP осуществляет процедуру т.н. «тройного рукопожатия» с сервером, оканчивающуюся установление соединения. Стоит отметить, что процедура рукопожатия никак не сказывается на работе приложения, т.к. выполняется прозрачно для него.

В ходе процесса установления соединения на серверной стороне создается новый сокет, сопоставленный с конкретным клиентским процессом.

С точки зрения приложения TCP-соединение является прямым ***дуплексным*** виртуальным каналом между сокетами соединения клиента и сервера. Например, клиент может осуществлять передачу любых байт через свой сокет, при этом протокол TCP гарантирует, что сервер получит эти байты через свой сокет без искажений и в том же порядке, в каком они были переданы. И что немаловажно в случае возникновения ошибок в среде передачи данных или разрыве соединения оба процесса будут уведомлены, что произошла ошибка при передаче данных.

Поскольку сокеты играют центральную роль в работе приложений клиент/серверных, разработку таких приложений часто называют программированием сокетов.

На рис. 1 приводится схема установления TCP-соединения и взаимодействия клиента и сервера.



**Рис. 1.** Взаимодействие TCP-сервера и клиента

С каждым сокет связываются три атрибута: домен, тип и протокол. Эти атрибуты задаются при создании сокета и остаются неизменными на протяжении всего времени его существования. Для создания сокета используется функция *socket*, имеющая следующий прототип.

int ***socket***(int domain, int type, int protocol);

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Чаще других используются домены Unix и Internet, задаваемые константами AF\_UNIX и AF\_INET соответственно (префикс AF означает «address family» – «семейство адресов»). При задании AF\_UNIX для передачи данных используется файловая система ввода/вывода Unix. В этом случае сокеты используются для межпроцессного взаимодействия на одном компьютере и не годятся для работы по сети. Константа AF\_INET соответствует Internet-домену. Сокеты, размещённые в этом домене, могут использоваться для работы в любой IP-сети. Существуют и другие домены (например, AF\_IPX для протоколов Novell, AF\_INET6 для новой модификации протокола IP - IPv6 и т. д.).

Тип сокета определяет способ передачи данных по сети. Чаще других применяются:

*SOCK\_STREAM*. Передача потока данных с предварительной установкой соединения. Обеспечивается надёжный канал передачи данных, при котором фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются. Этот тип сокетов является самым распространённым.

*SOCK\_DGRAM*. Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Обмен данными происходит быстрее, но является ненадёжным: сообщения могут теряться в пути, дублироваться и переупорядочиваться. Допускается передача сообщения нескольким получателям (multicasting) и широковещательная передача

(broadcasting).

*SOCK\_RAW*. Этот тип присваивается низкоуровневым (т. н. «сырым») сокетам. Их отличие от обычных сокетов состоит в том, что с их помощью программа может взять на себя формирование некоторых заголовков, добавляемых к сообщению.

Не все домены допускают задание произвольного типа сокета.

Например, совместно с доменом Unix используется только тип *SOCK\_STREAM*. В тоже время, для Internet-домена можно задавать любой из перечисленных типов. В этом случае для реализации *SOCK\_STREAM* используется протокол TCP, для реализации *SOCK\_DGRAM* – протокол UDP, а тип *SOCK\_RAW* используется для низкоуровневой работы с протоколами IP, ICMP и т. д.

Наконец, последний атрибут определяет протокол, используемый для передачи данных. Как мы только что видели, часто протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции socket можно передать 0, что соответствует протоколу по умолчанию. Тем не менее, иногда (например, при работе с низкоуровневыми сокетами) требуется задать протокол явно. Числовые идентификаторы протоколов зависят от выбранного домена; их можно найти в документации.

# Адреса

Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (эту процедуру называют именованием сокета). Иногда связывание осуществляется неявно (внутри функций connect и accept), но выполнять его необходимо во всех случаях. Вид адреса зависит от выбранного вами домена. В Internet-домене адрес задаётся комбинацией IP-адреса и 16-битного номера порта. IP-адрес определяет хост в сети, а порт - конкретный сокет на этом хосте. Протоколы TCP и UDP используют различные пространства портов.

Для явного связывания сокета с некоторым адресом используется функция *bind*. Её прототип имеет вид:

int bind(int sockfd, struct sockaddr \*addr, int addrlen);

В качестве первого параметра передаётся дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр, addr, содержит указатель на структуру с адресом, а третий – длину этой структуры. Структура выглядит следующим образом: struct sockaddr{

unsigned short sa\_family; // Семейство адресов, AF\_xxx char sa\_data[14]; // 14 байт для хранения адреса

};

Поле sa\_family содержит идентификатор домена, тот же, что и первый параметр функции *socket*. В зависимости от значения этого поля по-разному интерпретируется содержимое массива sa\_data. Разумеется, работать с этим массивом напрямую не очень удобно, поэтому вы можете использовать вместо sockaddr одну из альтернативных структур. Для Internet домена это sockaddr\_in. При передаче в функцию *bind* указатель на эту структуру нужно приводить к указателю на sockaddr. Рассмотрим для примера структуру sockaddr\_in.

struct sockaddr\_in {

short int sin\_family; // Семейство адресов unsigned short int sin\_port; // Номер порта struct in\_addr sin\_addr; // IP-адрес

unsigned char sin\_zero[8]; // "Дополнение" до размера

// структуры sockaddr

};

Здесь поле sin\_family соответствует полю sa\_family в sockaddr, в sin\_port записывается номер порта, а в sin\_addr – IP-адрес хоста. Поле sin\_addr само является структурой, которая имеет вид:

struct in\_addr { unsigned long s\_addr;

};

Для обратной совместимости (на уровне исходных кодов) со старыми приложениями единственное поле заключено в структуру, до этого же in\_addr представляла собой объединение (union), содержащее гораздо большее число полей.

# Сетевой порядок байт

Передача от одного вычислительного комплекса к другому символьной информации, как правило (когда один символ занимает один байт), не вызывает проблем. Однако для числовой информации ситуация усложняется.

Как известно, порядок байт в целых числах, представление которых занимает более одного байта, может быть для различных компьютеров неодинаковым. Есть вычислительные системы, в которых старший байт числа имеет меньший адрес, чем младший байт (big-endian byte order), а есть вычислительные системы, в которых старший байт числа имеет больший адрес, чем младший байт (little-endian byte order). Различие представлено в табл. 1. ***Таблица 1.*** Хранение целых чисел в памяти

**Число Последовательность**

**байт в памяти**

little-endian 0xAABBCCDD DD CC BB AA

big-endian AA BB CC DD

При передаче целой числовой информации от машины, имеющей один порядок байт, к машине с другим порядком байт мы можем неправильно истолковать принятую информацию. Для того чтобы этого не произошло, было введено понятие сетевого порядка байт, т.е. порядка байт, в котором должна представляться целая числовая информация в процессе передачи ее по сети (на самом деле – это big-endian byte order). Целые числовые данные из представления, принятого на компьютере-отправителе, переводятся пользовательским процессом в сетевой порядок байт, в таком виде путешествуют по сети и переводятся в нужный порядок байт на машине-получателе процессом, которому они предназначены. Для перевода целых чисел из машинного представления в сетевое и обратно используется четыре функции: htons(Host TO Network Short), htonl(Host TO Network Long), ntohs(Network TO Host Short), ntohl(Network TO Host Long).

При указании IP-адреса и номера порта необходимо преобразовать число из порядка хоста в сетевой (функции *htonl* и *htons* соответственно)

## Установка соединения (сервер)

Установка соединения на стороне сервера состоит из четырёх этапов, все они являются обязательными. Сначала сокет создаётся и привязывается к локальному адресу. Если компьютер имеет несколько сетевых интерфейсов с различными IP-адресами, вы можете принимать соединения только с одного из них, передав его адрес функции *bind*. Если же вы готовы соединяться с клиентами через любой интерфейс, задайте в качестве адреса константу INADDR\_ANY. Что касается номера порта, вы можете задать конкретный номер или 0 (в этом случае система сама выберет произвольный неиспользуемый в данный момент номер порта).

На следующем шаге создаётся очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов. Всё это выполняет функция *listen*.

int listen(int sockfd, int backlog);

Первый параметр – дескриптор сокета, а второй задаёт размер очереди запросов. Каждый раз, когда очередной клиент пытается соединиться с сервером, его запрос ставится в очередь, так как сервер может быть занят обработкой других запросов. Если очередь заполнена, все последующие запросы будут игнорироваться. Когда сервер готов обслужить очередной запрос, он использует функцию accept.

int accept(int sockfd, void \*addr, int \*addrlen);

Функция *accept* создаёт для общения с клиентом новый сокет и возвращает его дескриптор. Параметр sockfd задаёт слушающий сокет. После вызова он остаётся в слушающем состоянии и может принимать другие соединения. В структуру, на которую ссылается addr, записывается адрес сокета клиента, который установил соединение с сервером. В переменную, адресуемую указателем addrlen, изначально записывается размер структуры; функция accept записывает туда длину, которая реально была использована. Если вас не интересует адрес клиента, вы можете просто передать NULL в качестве второго и третьего параметров.

## Установка соединения (клиент)

На стороне клиента для установления соединения используется функция *connect*, которая имеет следующий прототип.

int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen);

Здесь sockfd – сокет, который будет использоваться для обмена данными с сервером, serv\_addr содержит указатель на структуру с адресом сервера, а addrlen – длину этой структуры. Обычно сокет не требуется предварительно привязывать к локальному адресу, так как функция connect сделает это за вас, подобрав подходящий свободный порт. Вы можете принудительно назначить клиентскому сокету некоторый номер порта, используя bind перед вызовом connect. Делать это следует в случае, когда сервер соединяется с только с клиентами, использующими определённый порт (однако такая ситуация чрезвычайно редка). В остальных случаях проще и надёжнее предоставить системе выбрать порт за вас.

## Обмен данными

После того как соединение установлено, можно начинать обмен данными. Для этого используются функции *send* и *recv*.

Функция *send* используется для отправки данных и имеет следующий прототип.

int send(int sockfd, const void \*msg, int len, int flags);

Здесь sockfd – это, как всегда, дескриптор сокета, через который мы отправляем данные, msg – указатель на буфер с данными, len – длина буфера в байтах, а flags – набор битовых флагов, управляющих работой функции (если флаги не используются, передайте функции 0). Вот некоторые из них (полный список можно найти в документации):

*MSG\_OOB*. Предписывает отправить данные как срочные (out of band data, OOB). Концепция срочных данных позволяет иметь два параллельных канала данных в одном соединении. Иногда это бывает удобно. Например, Telnet использует срочные данные для передачи команд типа Ctrl+C. В настоящее время использовать их не рекомендуется из-за проблем с совместимостью (существует два разных стандарта их использования, описанные в RFC793 и RFC1122). Безопаснее просто создать для срочных данных отдельное соединение.

*MSG\_DONTROUTE*. Запрещает маршрутизацию пакетов. Нижележащие транспортные слои могут проигнорировать этот флаг.

Функция *send* возвращает число байтов, которое на самом деле было отправлено (или -1 в случае ошибки). Это число может быть меньше указанного размера буфера. Если вы хотите отправить весь буфер целиком, вам придётся написать свою функцию и вызывать в ней *send*, пока все данные не будут отправлены. Она может выглядеть примерно так:

int sendall(int s, char \*buf, int len, int flags)

{

int total = 0; int n;

while(total < len)

{

n = send(s, buf+total, len-total, flags);

if(n == -1) break; total += n;

}

return (n==-1 ? -1 : total);

}

Использование *sendall* ничем не отличается от использования *send*, но она отправляет весь буфер с данными целиком. Для чтения данных из сокета используется функция *recv*. int recv(int sockfd, void \*buf, int len, int flags);

В целом её использование аналогично *send*. Она точно так же принимает дескриптор сокета, указатель на буфер и набор флагов. Флаг *MSG\_OOB* используется для приёма срочных данных, а *MSG\_PEEK* позволяет «подсмотреть» данные, полученные от удалённого хоста, не удаляя их из системного буфера (это означает, что при следующем обращении к *recv* вы получите те же самые данные). Полный список флагов можно найти в документации. По аналогии с *send* функция *recv* возвращает количество прочитанных байтов, которое может быть меньше размера буфера. Вы без труда сможете написать собственную функцию *recvall*, заполняющую буфер целиком. Существует ещё один особый случай, при котором recv возвращает 0. Это означает, что соединение было разорвано.

## Закрытие сокета

Закончив обмен данными, необходимо закрыть соединение и сам сокет. Это делается при помощи функции *closesocket*. Это приведёт к разрыву соединения.

int close (int fd);

## Принцип установления TCP-соединения на серверной стороне

Пара сокетов (sockets pair) для соединения TCP – это кортеж (группа взаимосвязанных элементов данных или записей) из 4-х элементов, определяющих две конечные точки соединения: локальный IP-адрес, локальный порт, удаленный IP-адрес и удаленный порт.

Два значения, идентифицирующих конечную точку, – IP-адрес и номер порта – часто называют TCP-сокетом (на самом деле понятие пары сокетов применимо и для протокола UDP).

Рассмотрим подробнее, что на самом деле происходит на стороне клиента и сервера при установлении TCP-соединения. Для наглядности рассмотрим поэтапно ситуацию установления соединения на простом примере.

Пускай у нас на машине с адресом 192.168.1.2 запущен Webсервер. Он ожидает подключение клиента на предопределенном 80-м порту. Эта ситуация изображена на рис. 2.

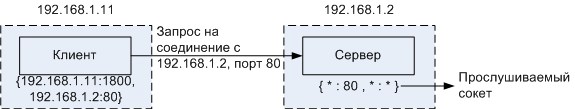


**Рис. 2.** Сервер с пассивным открытием на порте 80

Обозначение { \* : 80 , \* : \* } используется для указания пары сокетов сервера. Сервер ожидает запроса соединения на всех локальных интерфейсах (первая звездочка) на порт 80. Удаленный IPадрес и порт пока не известны, поэтому они обозначаются как «\* : \*».

Такая структура называются прослушивающим сокетом.

После запуска на узле 192.168.1.11 клиента и выполнения активного соединения с IP-адресом сервера 192.168.1.2 клиентскому сокету в данном примере назначается динамический порт 1800. Данная ситуация отображена на рис. 3.



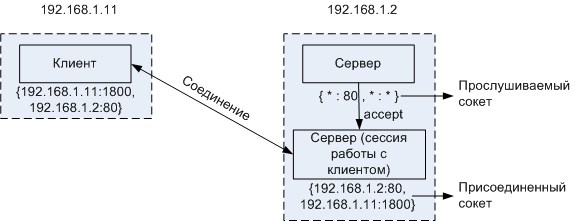
**Рис. 3.** Запрос на установление соединения от клиента 192.168.1.11

После принятия соединения на стороне сервера функция accept возвращает новую структуру сокета. Вновь созданный сокет носит название присоединенного. Обратите внимание, что как прослушивающий, так и присоединенный сокеты используют один и тот же локальный порт (80), также обратите внимание, что структура сокета теперь полностью заполнена – {192.168.1.2:80,

192.168.1.11:1800}.

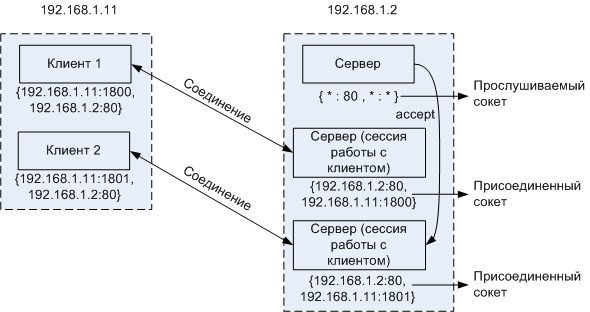
В дальнейшем вся работа по потоковой передачи байт осуществляется с использованием только присоединенного сокета

(см. рис. 4).



**Рис. 4.** Создание присоединенного сокета

После принятия соединения сервер может вновь перейти в режим прослушивания сокета (например, если программа многопоточная или используются возможности асинхронных сокетов), ожидая подключения новых клиентов. Ситуация работы сервера с 2-мя клиентами приведена на рис. 5.



**Рис. 5.** Работа с двумя клиентами одновременно

**Berkeley *Sockets API***

Ниже приведены прототипы соответствующих функций

Berkeley Sockets API на языке C.

#include <sys/types.h>

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

struct hostent \*gethostbyname(const char \*name);

struct hostent \*gethostbyaddr(const void \*addr, int len, int type); int connect(int sockfd, const struct sockaddr \*serv\_addr, socklen\_t addrlen); int bind(int sockfd, const struct sockaddr \*my\_addr, socklen\_t addrlen); int listen(int sockfd, int backlog);

int accept(int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen); int send(int sockfd, const void \*buf, size\_t len, int flags); ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

Задание на лабораторную работу

Необходимо создать клиент-серверное приложение, где клиентское приложение выполняет интерактивное взаимодействие с пользователем: ввод данных с клавиатуры и вывод результатов на экран. А серверное приложение получает от клиента набор данных для обработки и посылает результат обратно клиенту.

Передача данных между клиентом и сервером должна осуществляться с использованием сокетов TCP/IP. Для каждого подключения клиента сервер должен создавать отдельный поток, в котором будет выполняться обработка запроса пользователя. Из этого требования следует, что сервер должен корректно работать с несколькими подключенными клиентами. Логика чтения/записи данных из сокета должна корректно обрабатывать неполное чтение/запись из сокета (т.е. число прочитанных/записанных байт может отличаться от значения, указанного в параметре len), так же нужно корректно обрабатывать нулевой результат функции recv.

Данные, передаваемые через сокет, должны быть размещены в буфере приема/отправки в формате, оптимальном для обработки процессором, т.е. если в задании речь идет об обработке:

* целых чисел - тип данных должен быть целочисленных (int или long)
* вещественных чисел - тип данных float или double
* строковые значения – тип ASCIIZ-строка

Все входные значения для программы должны задаваться интерактивно (а не статически указанные значения в исходных кодах программы). Допустима/желательна реализация пользовательского интерфейса средствами консольного ввода-вывода. В случае если введенные данные являются недопустимыми, об этом необходимо информировать пользователя (разумеется, проверка на правильность данных должна выполняться в серверном приложении). После отображения результатов, пользователю необходимо дать возможность выхода из программы или переход к повторному вводу данных.

**Литература**

1. Таненбаум Э., Вудхал А. Операционные системы. Разработка и реализация. 4-е изд. – СПб.: Питер, 2015. – 704с.
2. Морис Бах. Архитектура Unix. <http://lib.ru/LINUXGUIDE/>
3. Иртегов Д.В. Введение в операционные системы. - СПБ: БХВ-Петербург, 2002.- 624с.