Лабораторная работа №3

Создание параллельного сервера с установлением логического соединения (TCP)

Цель работы – изучить методы создания серверов, используя алгоритм параллельной обработки запросов.

ПОВТОРЕНИЕ.

Взаимодействие с установлением логического соединения:

Наличие постоянного соединения (виртуального канала), существующего в стеке протоколв TCP/IP усложняет порядок взаимодействия сторон, но обеспечивает более развитый сервис, в первую очередь контроль целостности сеанса и передаваемых данных. Используемый тип сокета – *SOCK_STREAM*, которому в *IP*-сетях соответствует протокол *TCP*. Порядок взаимодействия схематично показан на рисунке 1.

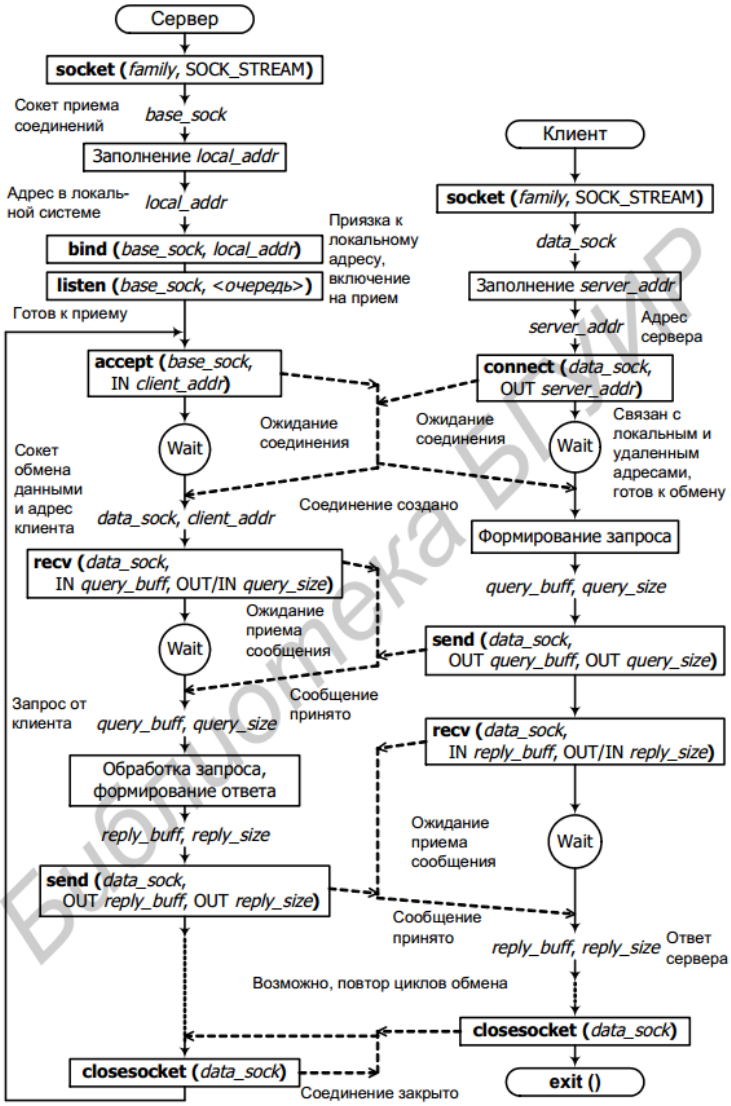

Рисунок 1 – Взаимодействие с установлением логического соединения

В начале работы сервер создает и связывает с локальным адресом **базовый сокет**, на котором он будет производить прием запросов на соединение. Именно порт, с которым он связан, указывается в качестве порта, занятого данным сервером. Далее базовый сокет переводится в **режим «прослушивания»**, и, начиная с этого момента, в системе организуется очередь, принимающая запросы на соединение. Основной цикл сервера состоит в приеме запросов на установление соединения вызовом функции *accept()*.

При поступлении запроса на установление соединения системой создается новое соединение, и вызов *accept()* на стороне сервера успешно завершается, возвращая дескриптор нового сокета, связанный с этим соединением – **сокет данных (сокет клиента)** готовый к использованию. Адрес для сокета данных выбирается системой автоматически, и он может совпадать с адресом базового сокета.

После этого сервер начинает вложенный цикл обмена данными через новое соединение, чаще всего это ожидание входящего запроса, его обработка и отправка ответа клиенту. В отличие от взаимодействия по протоколу UDP при передаче данных адрес назначения указывать не нужно, поэтому используется более простой вызов функции *send()*.

Признаком завершения обмена может служить, например, прием одной или нескольких порций данных нулевой длины. В протоколе прикладного уровня может быть также предусмотрены извещения от клиента о прекращении обмена или возможность разрыва соединения по инициативе сервера. После окончания вложенного цикла сервер может вернуться к приему следующего запроса на установление соединения.

Со своей стороны клиент создает сокет и передает его в вызов функции *connect()*, указывая также адрес сервера, с которым нужно установить соединение – *IP*-адрес хоста, где находится сервер, и закрепленный за ним порт. Делать предварительную привязку этого сокета к локальному адресу необязательно – в этом случае система выполнит ее автоматически.

После успешного завершения вызова *connect()* сокет будет связан с созданным соединением и готов к работе, клиент может начинать цикл обмена данными с сервером.

Блокирующими вызовами здесь помимо *recv()* являются *accept()* и *connect()*. Вызов *accept()* фактически принимает запрос на соединение из очереди, и если очередь пуста, то он остается в состоянии ожидания. Вызов *connect()* блокируется, если сгенерированный им запрос на соединение не обслужен сервером, но поставлен в очередь. Если в очереди сервера свободных мест нет, то запрос отвергается немедленно и соответствующий *connect()* клиента завершается как неуспешный.

Кроме того, во время обслуживания клиентского соединения сервер с описанным алгоритмом приостанавливает прием новых соединений (поэтому нет имеет смысла резервировать большую очередь для запросов). Размер очереди задается при «включении» базового сокета вызовом *listen()*.

Определенной проблемой является контроль целостности *TCP*-соединения. Алгоритмы *TCP* таковы, что прекращение связи обнаруживается только при обмене сегментами (система сама также может посылать специальные сегменты для проверки «активности» соединения). Корректный разрыв соединения также должен сопровождаться специальными флагами в заголовке сегмента. Поэтому полная потеря связи, аварийное (без соблюдения процедуры разрыва) завершение или просто неактивность программы на другом конце соединения без специальных мер обычно не отличимы друг от друга.

Вариантом решения этой проблемы может быть введение в прикладной протокол обмена (поверх *TCP*) обязательных периодических «пустых» посылок или специальных «зондирующих» сообщений в сочетании с использованием тайм-аутов.

ВЗАИМОДЕЙСТВИЕ БЕЗ УСТАНОВЛЕНИЯ ЛОГИЧЕСКОГО СОЕДИНЕНИЯ

Этот вид взаимодействия предусматривает обмен датаграммами, характеризуется минимальным уровнем сервиса со стороны системы и минимальными служебными затратами. Целостность передаваемых данных системой не гарантируются, при необходимости о ней должны заботиться сами взаимодействующие программы. Используемый тип сокета – *SOCK_DGRAM*, которому в *IP*-сетях соответствует протокол *UDP*. Порядок взаимодействия схематично показан на рисунке 2.

Полезной особенностью большинства протоколов датаграммной передачи, в том числе и *UDP*, является **широковещание (broadcasting)** – доставка соответствующим образом адресованного сообщения всем доступным узлам сети (подсети). Например, посредством широковещательного запроса можно обнаружить сервер, адрес которого неизвестен (но порт программы-сервера знать необходимо), для чего ряд служб, ориентированных на использование *TCP*-соединений, поддерживают также и интерфейс *UDP* (широковещание действует не всегда: оно может не поддерживаться, быть запрещено административно или отключаться при отсутствии реального соединения с сетью, когда функционирует только интерфейс *localhost*).

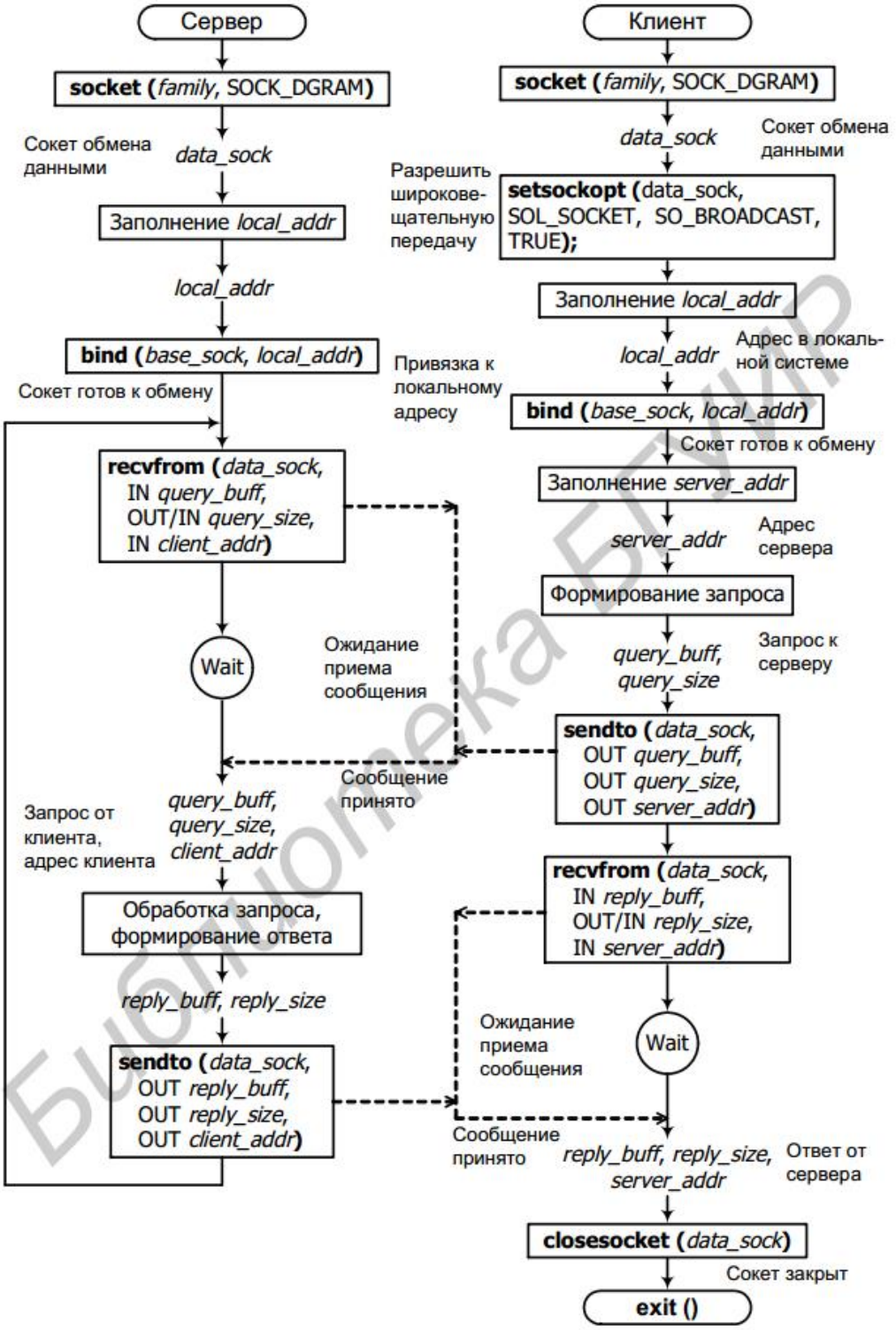

Рисунок 2 – Взаимодействие без установления логического соединения

На рисунке имена системных функций сохранены, но формат их вызова упрощен для повышения наглядности. Аргументы, описывающие передаваемые и принимаемые сообщения, снабжены пометками, являются ли соответствующие объекты параметрами вызова (*IN*) или создаются (заполняются) в результате его выполнения (*OUT*).

В показанном примере действия клиента и сервера в общем схожи, и с точки зрения программирования сокетов отличия между ними только

номинальные. Каждый из них использует всего один сокет, служащий и для приема и для передачи сообщений. Для упрощения будем считать, что клиент отсылает серверу единственной запрос (*query*) и ожидает ответ на него (*reply*), а сервер в цикле принимает запросы, обрабатывает их и отсылает ответы, используя «обратные адреса» запросов. Цикл сервера показан бесконечным, но на практике для управления серверами служат команды, предусмотренные в протоколе обмена, или какие-либо дополнительные средства.

Важно отметить, что некоторые действия в программах являются блокирующими, то есть приостанавливают выполнение на неопределенный срок. В первую очередь, это ожидание поступления сообщения (функция *recvfrom()*). Подобная ситуация характерна для сетевых приложений, причем она усугубляется неопределенностью состояния партнера и каналов связи. В результате, например, при отсутствии сервера может произойти как распознавание ошибки приема, так и переход клиента в состояние бесконечного ожидания (часто этот эффект наблюдается только при запросах на широковещательный адрес), соответствие пар запрос-ответ может нарушаться, если возвращается более одного ответа и т. д. Для реальных приложений такие дефекты обычно недопустимы.

Особенности параллельного соединения

В предыдущих лабораторных работах были показаны примеры применения транспортного протокола с установлением логического соединения в **последовательном** сервере, который способен взаимодействовать с несколькими клиентами поочередно, то есть новое соединение не создается ранее закрытия текущего соединения, новый запрос не принимается ранее окончания обработки текущего. Такой тип сервера характеризуется простотой и минимальными затратами ресурсов на служебные функции, однако невозможность работы более чем с одним клиентом одновременно остается принципиальным ограничением. Следовательно, последовательный сервер применим в случаях, когда заведомо не требуется обслуживать более чем одного клиента или интенсивность запросов очень мала, или запросы можно выполнять очень быстро, причем соединение (если используется *TCP*) затем немедленно разрывается самим сервером (примером последнего может служить служба *daytime*).

В лабораторной работе №3 будет приведен пример параллельного сервера, в котором используется транспортный протокол с установление логического соединения.

Большинство серверов должно иметь дело со многими клиентами, присылающими, возможно, очень большой поток запросов. При этом помимо эффективного распределения ресурсов сервера, необходимо решить проблему совмещения во времени ряда операций:

– создание новых соединений;

– прием данных (запросов) от каждого из клиентов;

– обработка каждого из принятых запросов;

– отсылка ответов на запросы;

– интерфейсные функции программы-сервера: прием и обработка сообщений, ввод с консоли и т. п.;

– другие, не связанные с сетью обращения к системе.

Таким образом, выполнение одновременно нескольких действий может понадобиться даже при обслуживании сервером единственного клиента. Большинство этих действий так или иначе связано с вводом-выводом и, следовательно, сопровождаются переходом в состояние ожидания, которое в случае простого последовательного сервера блокирует прочую активность.

В частности, при обращении к сокетам, в первую очередь за получением некоторых данных, причинами неопределенно длительного ожидания могут стать:

– задержки передачи через сеть, особенно глобальную;

– особенности функционирования модулей транспортной системы;

– асинхронность и случайный характер происходящих событий;

– непредсказуемые ошибки;

– неопределенность состояния противоположной стороны.

В программе все они проявятся как ожидание завершения некоторых функций, основными из которых будут:

– accept() – установление соединений, «внешний» цикл TCP-сервера;

– connect() – установление соединения на стороне TCP-клиента;

– recv() и recvfrom() – прием данных, как TCP, так и UDP;

– send() и sendto() – передача данных (как источник блокировки эти функции рассматриваются гораздо реже).

Легко обнаружить, что именно ожидание будет составлять значительную долю расходуемого сервером времени, а благодаря асинхронности событий, можно совместить ожидание, связанное с одними клиентами, с обслуживанием активности других. Следовательно, сервер сможет поддерживать общение более чем с одним клиентом так, что время отклика для каждого из них будет оставаться в приемлемых пределах – конечно, если сервер не перегружен запросами. Такой сервер будем называть *многопользовательским* (для большинства сетевых служб естественен именно такой режим работы сервера).

Примечание. Большая часть материала относится к случаю TCP-сервера как наиболее характерному (обычно необходимо поддерживать несколько клиентских соединений). Однако актуальность его сохраняется и для UDP-сервера (выполнение длительных запросов на фоне приема следующих) и даже для клиентских программ (интерфейс с сетью и одновременно с пользователем).

Использование системной многозадачности

Наиболее очевидным решением является вынесение цепочек логически связанных потенциально блокирующих операций в отдельные ветви алгоритма, выполняющиеся независимо от остальных. Такими цепочками могут стать:

– цикл приема соединений;
– циклы обслуживания соединений;
– выполнение отдельных запросов;
– при необходимости – интерфейсные функции.
Упрощенная схема такого сервера показана на рисунке 3.

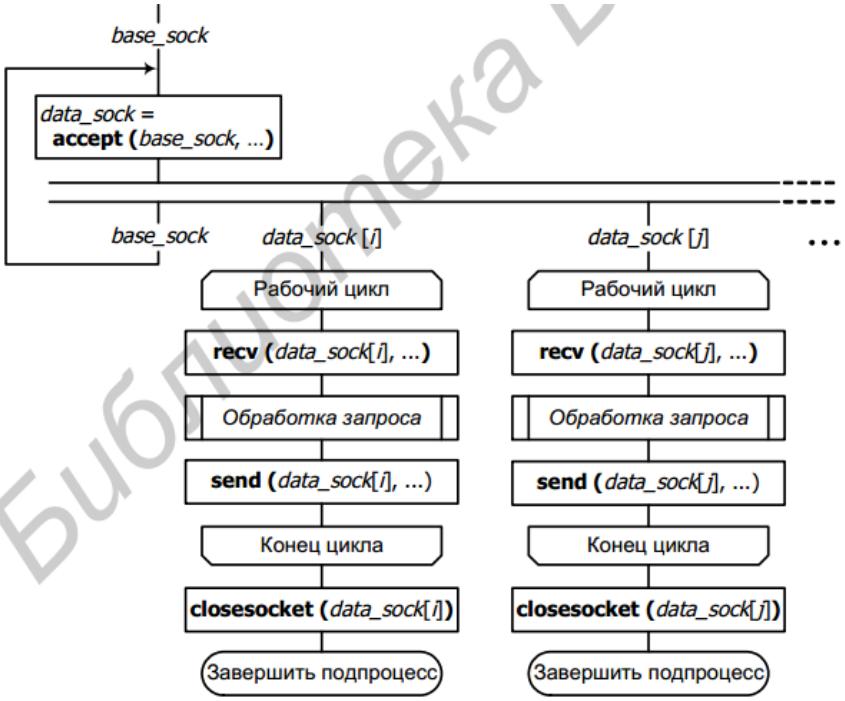

Рисунок 3 – Упрощенная схема сервер с использованием системной многозадачности

Здесь «главный» сокет (base_sock) принимает запросы на соединение, после чего порождает новый экземпляр «исполнительного» сокета (data_sock), который будет заниматься обслуживанием этого соединения, а «главный» сокет возвращается к ожиданию следующего запроса на установление соединения. Все порожденные экземпляры (base_sock[i]) функционируют и завершаются независимо друг от друга. Таким образом, за распараллеливание операций отвечают системные средства обеспечения многозадачности, присутствующие в большинстве современных операционных систем.

На рисунке 3 эти экземпляры названы «подпроцессами», на практике их роль в зависимости от особенностей конкретной системы могут играть как собственно процессы (process), так и потоки (thread). Использование этих двух методик зависит от требуемой ресурсоемкости и от эффективных способов создания новых процессов, существующих в операционной системе. В *Win* 32 для создания процессов и потоков служат вызовы функций *CreateProcess()* и *CreateThread()* соответственно.

К достоинствам многопотокового подхода можно отнести:

– относительная простота и «прозрачность» распараллеливания;

– высокая эффективность и хорошая масштабируемость при выполнении на параллельной (многопроцессорной) ЭВМ.

Недостатками являются значительные затраты на поддержание процессов (потоков) и сложности при их взаимодействии между собой. При достаточно большом количестве одновременно выполняющихся процессов (потоков) управляющие ими механизмы операционной системы перегружаются: начинается «пробуксовка» планировщика, возрастают затраты времени на переключение контакта и потери из-за конфликтов. Задача синхронизации в программах с элементами параллелизма и особенно отладка таких программ также очень сложны и не всегда имеют однозначное эффективное решение.

Переключение контакта – 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.

Основное различие между потоком и процессом заключается в том, что все потоки одного процесса выполняются в общем адресном пространстве и могут использовать общие глобальные переменные, что упрощает взаимодействие между ними (но не снимает проблему синхронизации!). Потоки потенциально менее ресурсоемки, хотя реализация их в различных конкретных системах может сильно различаться. Операционные системы семейства Win 32 изначально многопоточные, более того, планировщик в них *всегда* работает именно с потоками, а не с процессами, поэтому использование потоков в Win 32 оказывается более эффективным и используется чаще. С другой стороны, процессы менее зависимы друг от друга, и зависание или аварийное завершение одного из них (не главного) скорее всего не повлияет на остальных, поэтому многопроцессный вариант в целом более надежен и устойчив. Так же поток порождается внутри процесса из одной из его подпрограмм, а процесс – из внешнего исполняемого файла либо путем клонирования исходного родительского процесса.

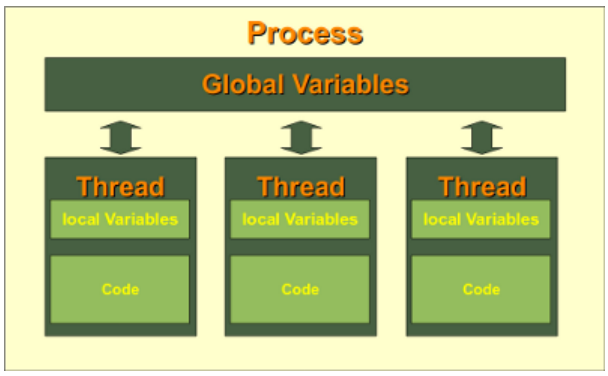

Рисунок 4 – Схематическое отображение понятия глобальных переменных

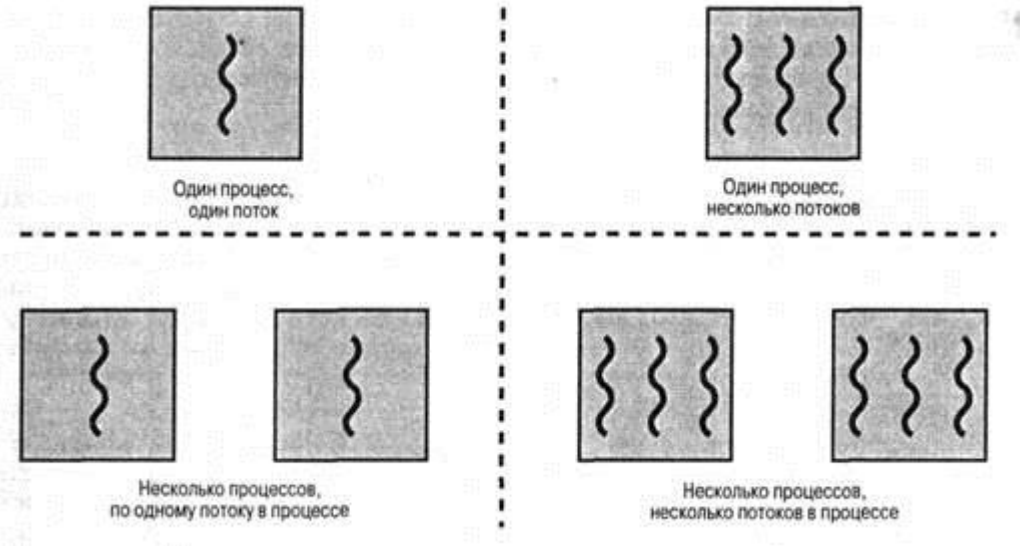

Рисунок 5 – Схема организации процессов и потоков

В чем разница между потоком и процессом?

Процессы и потоки связаны друг с другом, но при этом имеют существенные различия. Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

Поток — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса. У каждого потока есть

собственные регистры и собственный стек, но другие потоки этого процесса могут их использовать.

Регистры *– специальные ячейки памяти, расположенные непосредственно в процессоре. Работа с регистрами выполняется намного быстрее, чем с ячейками оперативной памяти, поэтому регистры активно используются как в программах на языке ассемблера, так и компиляторами языков высокого уровня.*

Второе преимущество потоков, т.е. разделяемая память. 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. Например, рассмотрим библиотечную функцию *gethostbyname*(), используемую в приложениях для преобразования доменного имени в *IP*-адрес. Если два потока вызовут функцию *gethostbyname*() одновременно, то ответ на один поисковый запрос может быть перекрыт ответом на другой. Поэтому если несколько потоков вызывают определенную библиотечную функцию, они должны координировать свою работу для обеспечения того, чтобы в любое время ее вызов выполнял только один поток.

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.

Так как особенно много ресурсов расходуется на само порождение процесса (потока), то существует необходимость минимизировать эти

расходы. В результате получается разновидность сервера с *предварительно порожденными* (*preforked*, *pre-created*) процессами (потоками), выполняющими функции «исполнительных» серверов. Изначально все они находятся в приостановленном (ждущем, *suspended*) состоянии. При появлении запроса главный цикл активирует одного из них для обслуживания клиента, а после завершения обработки «исполнительный» сервер снова становится неактивным. Их количество выбирается заранее с таким расчетом, чтобы иметь возможность удовлетворять достаточно много запросов, но не перегружать систему. Так же переключение «исполнительных» серверов может выполняться иначе: вместо главного цикла все одинаковые исполнительные серверы сами обращаются за получением запроса, переходя в состояние ожидания, и активизация одного из них выполняется уже непосредственно системой. С той же целью можно использовать **семафоры** или другие **объекты ожидания**. **Семафор (semaphore)** — объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Определение введено Эдсгером Дейкстрой. Семафоры используются для синхронизации и защиты передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков.

В целом многозадачный вариант для сетевых приложений наиболее эффективен в случаях, когда число одновременно обслуживаемых клиентов ограничено, интенсивность установления новых соединений и поступления запросов велики, сами запросы достаточно сложны и требуют длительной обработки, а взаимодействие между «исполнительными» элементами сервера минимально (то есть синхронизация требуется редко).

Сервер при обеспечении параллельного формирования ответов на запросы опирается на поддержку параллельного выполнения процессов операционной системой. Системный администратор предусматривает автоматический запуск ведущего серверного процесса во время начальной загрузки системы. Ведущий сервер функционирует неопределенно долгое время, ожидая поступления новых запросов на установление соединения от клиентов. Ведущий поток создает новый ведомый поток для обработки запросов каждого нового соединения и предоставляет каждому ведомому потоку возможность взять на себя весь обмен данными с клиентом.

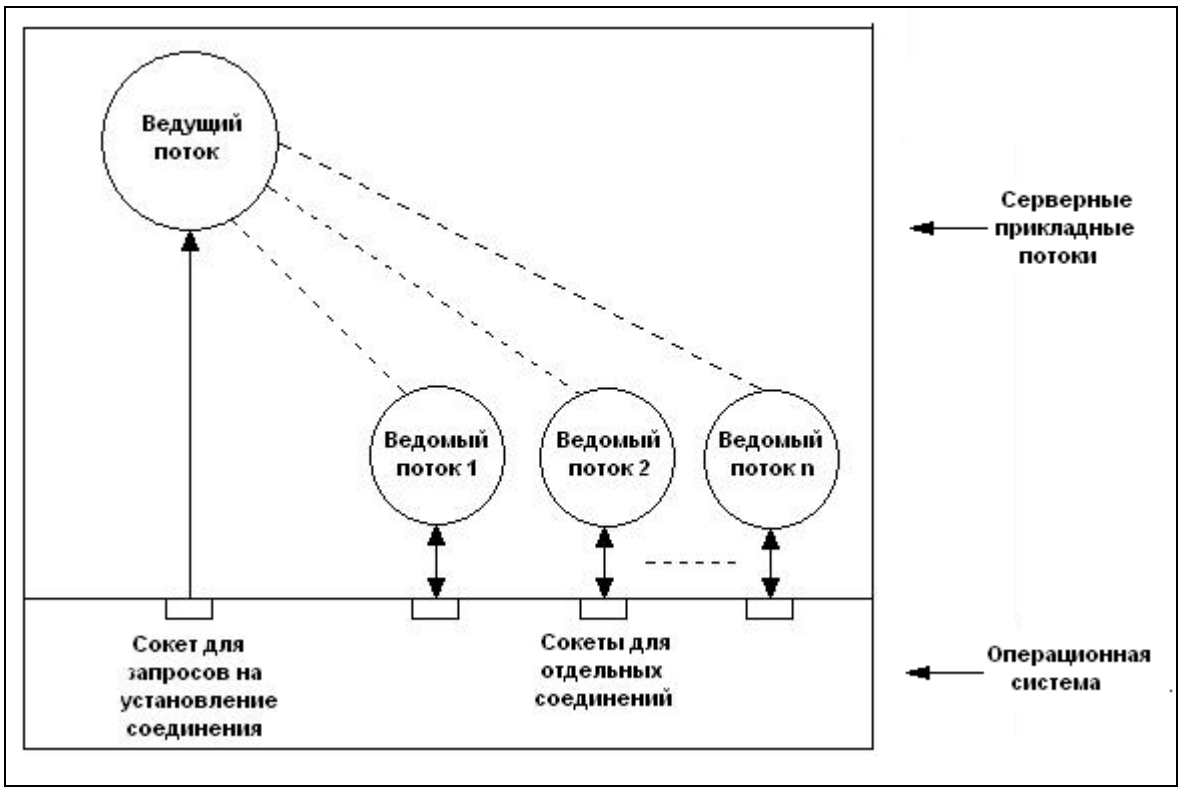

Рисунок 6 – Схема организации процессов параллельного сервера с установлением логического соединения, в котором используются однопотоковые процессы

Чтобы дать возможность клиенту передавать произвольные объемы данных, сервер не выполняет чтение всей входной информации до начала отправки ответа. Вместо этого сервер чередует прием и передачу. После поступления нового запроса на установление соединения сервер входит в цикл. При каждом проходе по циклу сервер сначала читает данные, поступающие из соединения, а затем снова пишет эти данные в то же соединение. Сервер повторно выполняет эти операции до тех пор, пока не встретит признак конца файла, после чего он закрывает соединение.

Сравнение последовательных и параллельных реализаций.

– последовательная реализация сервера может оказаться неудовлетворительной, 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, т.к. не позволяет одному клиенту захватить все ресурсы. Вместо этого параллельный сервер поддерживает обмен данными сразу с несколькими клиентами для того, чтобы их запросы выполнялись одновременно. Поэтому, с точки зрения клиента, параллельный сервер

обеспечивает лучшее наблюдаемое время отклика по сравнению с последовательным сервером.

3.2. Методические указания

Рассмотрим многопоточность на следующем примере.

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 4, то первая часть строки меняется местами со второй. Результаты преобразований возвращаются назад клиенту.

Как и в предыдущих лабораторных работах будем использовать разработку приложений типа клиент/сервер.

Серверная часть:

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

Функция *CreateThread()* создает поток, который выполняться 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 в байтах
LPTHREAD_START_ROUTINE lpStartAddress,//указатель на функцию потока
LPVOID lpParameter, //адрес параметра для нового потока
DWORD dwCreationFlags, //флаги создания потока
LPDWORD lpThreadId // идентификатор потока
 );
```

Параметр *lpThreadAttributes* устанавливает атрибуты защиты создаваемого потока и 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.

Параметр *dwStackSize* определяет начальный размер стека в байтах, который выделяется потоку при запуске. Система округляет это значение до ближайшей единицы. 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. Операционная

система Windows округляет размер стека до одной страницы памяти, который обычно равен 4 Кб.

Параметр *lpStartAddress* – указатель на определенную прикладную функцию типа *LPTHREAD_START_ROUTINE*, для выполнения ее потоком и представления начального адреса потока. Эта функция должна иметь следующий прототип:

```
DWORD WINAPI ThreadProc (LPVOID lpParameters);
```

Параметр *lpParameter* является единственным параметром, который будет передан функции потока.

Параметр *dwCreationFlags* используется для определения дополнительных флагов, которые управляют созданием потока. Если определен флаг *CREATE_SUSPENDED,* то поток будет создан в приостановленном (подвешенном) состоянии и не будет работать, пока функция *ResumeThread()* не будет вызвана. Если значение этого параметра равно 0, то поток выполняется немедленно после его создания. В то же время, никакие другие величины не предусмотрены.

Параметр *lpThreadId* является выходным и его значение устанавливает Windows. Этот параметр должен указывать на переменную, в которую Windows поместит идентификатор потока, который уникален для всей системы и может в дальнейшем использоваться для ссылок на поток.

Если функция *CreateThread()* успешно выполняется, то возвращаемое значение есть дескриптор созданного потока, который является уникальным для всей системы. В случае невыполнения функции, возвращаемое значение принимает значение *NULL*. Для получения большей информации об ошибке, можно обратиться к функции *GetLastError().*

Примечание:

✓ 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 не доступен. 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 мы определим сами следующим образом:

```
DWORD WINAPI ThreadFunc(LPVOID client_socket){
	cout << "Thread is started." << endl;
	// тип LPVOID - указатель на любой тип
	SOCKET s2 =((SOCKET *) client_socket)[0];
	char buf[100], buf1[100];
	send(s2,"Welcome new client!\n",sizeof("Welcome new
	client!\n"),0);
	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);
	cout << "Thread is finished." << endl;
	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 is ready to receive message"<<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 окончанию работы функция *CreateThread()* закрывает поток, инициализирует используемые указатели значением NULL. Существует специальная функция *ExitThread()*, выполняющая аналогичные действия. Её прототип:

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

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

Примечание:

✓ 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 приложения клиента предыдущих лабораторных работ. Согласно требованиям условия задачи клиентская часть имеет следующий вид:

```
#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<<"The string received:"<<endl<<buf<<endl;
                }
                closesocket(s);
                }
        WSACleanup();
}
```

Указатели на функции.

Функция располагается в памяти по определённому адресу, который можно присвоить указателю. Адресом функции является точка её входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Указатель на функцию позволяет также передавать ее другим функциям в качестве аргумента. В программах на языке C/C++ адресом функции служит ее имя без скобок и аргументов. Рассмотрим программу, сравнивающую две строки, введенные пользователем.

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,int (*cmp)(const char *, const char *));
int main(void){
        char s1[80], s2[80];
        int (*p)(const char *, const char *);/* указатель на функцию */
        p = strcmp; /* присваивает адрес функции strcmp указателю p */
        printf("Введите две строки.\n");
        gets(s1);
        gets(s2);
        check(s1,s2,p);//Передает адрес функции посредством указателя p
        return 0;
}
void check(char *a, char *b,int (*cmp)(const char *, const char *)){
        printf("Проверка на совпадение.\n");
        if(!(*cmp)(a, b)) printf("Равны");
        else printf("Не равны");
}
```

Рассмотрим объявление указателя *p* в функции *main()*:

```
int (*p)(const char *, const char *);
```

Это объявление сообщает компилятору, что *p* — это указатель на функцию, имеющую два параметра типа const char * и возвращающую значение типа int. Скобки вокруг *p* необходимы для правильной интерпретации объявления компилятором. Подобная форма объявления используется также для указателей на любые другие функции, нужно лишь

внести изменения в зависимости от возвращаемого типа и параметров функции.

В функции *check()* объявлены три параметра: два указателя на символьный тип (*a* и *b*) и указатель на функцию *cmp*. Этот указатель функции *cmp* объявлен в том же формате, что и указатель *p*. Поэтому в *cmp* можно хранить значение указателя на функцию, имеющую два параметра типа *const char ** и возвращающую значение *int*. Как и в объявлении *p*, круглые скобки вокруг **cmp* необходимы для правильной интерпретации этого объявления компилятором.

В программе указателю *p* присваивается адрес стандартной библиотечной функции *strcmp()*, которая сравнивает строки. Далее программа просит пользователя ввести две строки и передает указатели на них функции *check()*, которая их сравнивает. Внутри *check()* выражение

```
(*cmp)(a, b)
```

вызывает функцию *strcmp()*, на которую указывает *cmp*, с аргументами *a* и *b*. Скобки вокруг **cmp* обязательны. Существует и другой, более простой, способ вызова функции с помощью указателя:

```
cmp(a, b);
```

Однако первый способ используется чаще и его рекомендуется использовать, потому что при втором способе вызова указатель *cmp* очень похож на имя функции, что может сбить с толку читающего программу. В то же время у первого способа записи есть свои преимущества, например, хорошо видно, что функция вызывается с помощью указателя на функцию, а не имени функции. Следует отметить, что первоначально в C был определен именно первый способ вызова.

Вызов функции *check()* можно записать, используя непосредственно имя *strcmp()*:

```
check(s1, s2, strcmp);
```

В этом случае вводить в программу дополнительный указатель *p* нет необходимости.

Во многих случаях оказывается более выгодным передать имя функции как параметр или даже создать массив функций. Например, в программе интерпретатора синтаксический анализатор (программа, анализирующая выражения) часто вызывает различные вспомогательные функции, такие как вычисление математических функций, процедуры ввода-вывода и т.п. В таких случаях чаще всего создают список функций и вызывают их с помощью индексов.

Альтернативный подход — использование оператора *switch* с длинным списком меток *case* — делает программу более громоздкой и подверженной ошибкам.

В следующем примере рассматривается расширенная версия предыдущей программы. В этой версии функция *check()* устроена так, что может выполнять разные операции над строками *s1* и *s2* (например, сравнивать каждый символ с соответствующим символом другой строки или сравнивать числа, записанные в строках) в зависимости от того, какая функция указана в списке аргументов. Например, строки "0123" и "123" отличаются, однако представляют одно и то же числовое значение.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void check(char *a, char *b,int (*cmp)(const char *, const char *));
int compvalues(const char *a, const char *b);
int main(void){
	char s1[80], s2[80];
	printf("Введите два значения или две строки.\n");
	gets(s1);
	gets(s2);
	if(isdigit(*s1)) {
		printf("Проверка значений на равенство.\n");
		check(s1, s2, compvalues);
		}
	else {
		printf("Проверка строк на равенство.\n");
		check(s1, s2, strcmp);
	}
	return 0;
}
void check(char *a, char *b, int (*cmp)(const char *, const char *)){
	if(!(*cmp)(a, b)) printf("Равны");
	else printf("Не равны");
}
int compvalues(const char *a, const char *b){
	if(atoi(a)==atoi(b)) return 0;
	else return 1;
}
```

Если в этом примере ввести первый символ первой строки как цифру, то check() использует compvalues(), в противном случае — strcmp(). Функция check() вызывает ту функцию, имя которой указано в списке аргументов при вызове check(), поэтому она в разных ситуациях может вызывать разные функции. Ниже приведены результаты работы этой программы в двух случаях:

```
Введите два значения или две строки.
тест
тест
Проверка строк на равенство.
Равны
Введите два значения или две строки.
0123
```

```
123
Проверка значений на равенство.
Равны
```

Сравнение строк 0123 и 123 показывает равенство их значений. Обратите внимание, что в языке С нулем начинаются восьмеричные константы. Если бы эта запись была в выражении, то 0123 не было бы равно 123. Однако здесь функция *atoi()* обрабатывает это число как десятичное.

Контрольные вопросы.

1. Что такое параллельное соединение? Особенности параллельного соединения.
2. Отличия параллельного соединения и последовательного?
3. Назовите преимущества многопотоковых процессов по сравнению с однопотоковыми процессами.
4. С чем связано повышение эффективности многопотоковых процессов?
5. Что позволяет контролирующему потоку формировать отчеты об активности ведомых потоков сервера для системного администратора?
6. Назовите недостатки многопотокового процессора по сравнению с однопотоковым.
7. Какие функции служат для создания потоков?

Варианты индивидуального задания

1. 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 3, то удаляются все числа, которые делятся на 3. Результаты преобразований этой строки возвращаются назад клиенту.
2. 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 четная, то удаляются 3 первых и 2 последних символа. Результаты преобразований этой строки возвращаются назад клиенту.

3. 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. Функциональные возможности сервера реализовать следующим образом: 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. 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 нечетная, то удаляется символ, стоящий посередине строки. Преобразованная строка передается назад клиенту.

5. 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 заменить в этой строке каждый второй символ @ на #. Результаты преобразований передаются назад клиенту.

6. 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 заменить в этой строке символов все пробелы на символ *. Преобразованная строка передается назад клиенту.

7. 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 длина больше 15, то из нее удаляются все цифры. Клиент получает преобразованную строку и количество удаленных цифр.

8. 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 длина кратна 4, то удаляются все числа, делящиеся на 4. Клиент получает преобразованную строку и количество таких чисел.

9. 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 5, то подсчитывается количество скобок всех видов. Их количество посылается клиенту.

10. 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 4, то первая часть строки меняется местами со второй. Результаты преобразований возвращаются назад клиенту.

11. 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 значение этой длины равно 10, то удаляются все символы от A до Z. Результаты преобразований такой строки и количество удалений возвращаются назад клиенту.

12. 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 длина больше 15, то удаляются все символы от a до z.

Преобразованная строка и количество удаленных символов возвращаются назад клиенту.

13. 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 в полученной строке символов поменять местами символы на четных и нечетных позициях. Полученную строку возвратить назад клиенту.

14. 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 длина больше 7, то выделяется подстрока в {} скобках и возвращается назад клиенту.

15. 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 длина больше 15, то выделяется подстрока до первого пробела и возвращается назад клиенту.