# Лабораторная работа

**Тема**

Разработка многопоточного сетевого сервера

**Цель работы**

Получить практические навыки разработки многопоточного сетевого сервера.

**Время выполнения работы**

2 академических часа

**Используемое методическое и лабораторное обеспечение**

1. 2 виртуальные машины с установленными дистрибутивами Debian.
2. Настроенная сеть между ними (192.168.111.10 для сервера и 192.168.111.11 для клиента)
3. ПК с текстовым редактором (для подготовки отчета).

# Теоретические сведения

## Общие сведения

Для исполнения программ используются объекты, называемые процессами. Процессу выделяется собственное виртуальное адресное пространство и некоторые иные ресурсы. Для каждого процесса создается иллюзия последовательного исполнения. Программа исполняется в рамках процесса так же, как она исполнялась бы в однозадачной ОС.

Процесс взаимодействует с ядром ОС при помощи системных вызовов. При исполнении системного вызова, процесс исполняет специальную команду, которая переключает адресное пространство и передает управление ядру. Процессы в традиционных Unix-системах могут взаимодействовать друг с другом только при помощи системных вызовов – чтения и записи в каналы и разделяемые файлы, в иных системах – при помощи System V IPC.

Таким образом, процессы надежно изолированы друг от друга. Нарушения целостности данных одного процесса приводят к аварийному завершению этого процесса, но не затрагивают напрямую другие процессы. Даже удаленное исполнение кода в рамках одного из процессов позволяет злоумышленнику, написавшему код, получить лишь те привилегии, которые имел данный процесс.

Однако ряд задач требует реализации в виде нескольких параллельно (или квазипараллельно) исполняющихся потоков.

**2. Преимущества применения многопоточности**

1. Улучшение времени реакции интерактивных программ.
2. Улучшение времени реакции серверных приложений. Возможность обрабатывать несколько запросов одновременно.
3. Использование дополнительных ресурсов на многопроцессорных и гипертрединговых компьютерах.
4. Задачи реального времени.

Следует подобротнее рассмотреть второй пункт.

Основные две характеристики серверов приложений – это среднее время исполнения запроса и количество запросов, обрабатываемых в единицу времени. Причем, это две разные характеристики.

Среднее время исполнения запроса – это основная характеристика производительности сервера с точки зрения отдельного пользователя, а конкретнее со стороны клиентского приложения. Увеличение этого времени ведет к снижению производительности пользователя при работе с сервером.

Количество запросов, обрабатываемых в единицу времени – это основная характеристика производительности сервера с точки зрения владельца этого сервера, то есть непосредственно со стороны серверного приложения. Нахождение способа увеличить данное количество, не ухудшая или незначительно ухудшая показатели исполнения каждого из запросов, позволит произвести подключение большего количества пользователей к серверу.

Процесс исполнения одиночного запроса типичным серверным приложением описан на рис. 1. Он состоит из четырех этапов: приема запроса, выполнение запроса, создания ответа и передачи ответа. Соответственно, время исполнения каждого из этих этапов может различаться в зависимости от типа сервиса и от самих запросов. Рассмотрим теоретический сервер, обрабатывающий однотипные запросы, у которых структура обработки запроса такова, как представлена на рис. 1, а время исполнения всех этапов относительно одинаково.

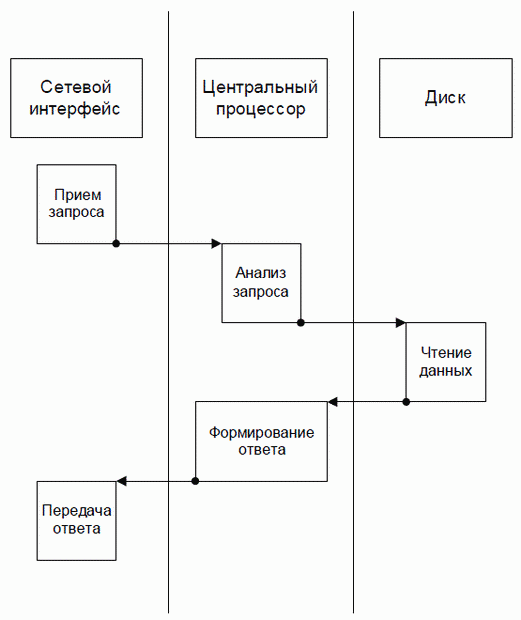


Рисунок - Исполнение одиночного запроса

Рассмотрим исполнение потока запросов. Однопоточный сервер должен исполнять запросы последовательно, поэтому максимальное количество запросов, исполняемых в секунду, равно 1/t, где t – время исполнения одиночного запроса.

При этом среднее время исполнения запроса не будет равно t, а будет расти в зависимости от вероятности перекрытия запросов во времени. Теоретические расчеты показывают, что, когда поток запросов приближается к 1/t в секунду, эта вероятность становится весьма значительной, так что время исполнения одиночного запроса может увеличиться во несколько раз. Однако к многопоточному серверу эти расчеты неприменимы. На рис.2 представлена схема обработки потока запросов многопоточным сервером.

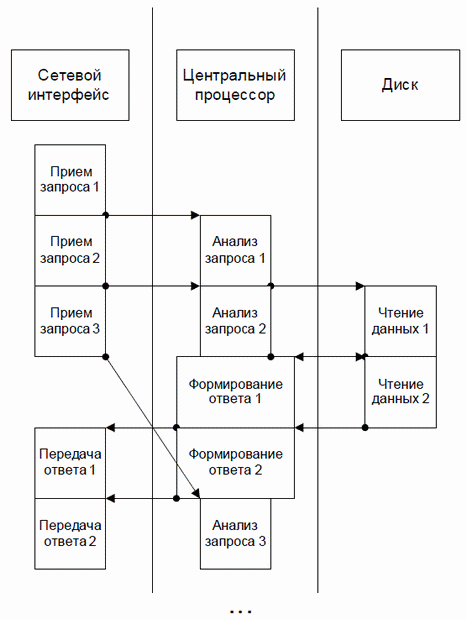


Рисунок - Обработка потока запросов многопоточным сервером

Видно, что сервер совмещает этапы исполнения перекрывающихся во времени запросов. Если относительные времена исполнения этапов запросов таковы, как на рис.1 и рис.2, избежать взаимодействия запросов не удается, так что среднее время исполнения запроса может вырасти по сравнению с t. Но количество запросов, исполняемых в секунду, оказывается значительно больше, чем 1/t.

Более того, очевидно, чтобы повысить производительность системы, можно установить второй процессор или второй сетевой интерфейс для повышения пропускной способности сети.

Однако реорганизация однопоточного серверного приложения в многопоточное требует специальной поддержки со стороны операционной системы и значительной переработки кода приложения.

**3. Недостатки многопоточности**

Первые попытки организации параллельных вычислений предпринимались еще в 60е годы XX столетия. Теория межпроцессного и многопоточного взаимодействия была в основном разработана еще тогда, однако массовое распространение многопоточное программирование получило лишь в 90е годы. Это было обусловлено рядом причин, основные из которых перечислены ниже:

1. Несовместимость со старыми (однопоточными) компиляторами
2. Несовместимость со старыми библиотеками
3. Несовместимость или ограниченная поддержка многопоточных программ другими инструментальными средствами, в первую очередь отладчиками
4. Несовместимость многих принятых практик программирования с многопоточностью.

Первые три из названных причин чисто технические, для их решения требуется переработка инструментальных средств разработки программ. Однако, поскольку такая переработка нарушает совместимость, массовый переход на новый инструментарий занял многие годы. К тому же, поскольку первые версии многопоточных инструментальных средств обладали различными недостатками, это также затрудняло их принятие разработчиками.

## Описание используемых библиотек

### threading

Стандартная библиотека Python предоставляет библиотеку threading, которая содержит необходимые классы для работы с потоками. Основной класс в этой библиотеки Thread.

Чтобы запустить отдельный поток, нужно создать экземпляр потока Thread и затем запустить его с помощью метода .start(). Когда вы создаете поток **Thread**, вы передаете ему функцию и список, содержащий аргументы этой функции.

MyThread = threading.Thread(target=function, args=(arg1,arg2,…))

MyThread.start()

Чтобы **указать одному потоку дождаться завершения другого потока,** вам нужно вызывать **.join()**.

MyThread.join()

threading.active\_count()Эта функция возвращает количество исполняемых на текущий момент потоков.

threading.current\_thread() Эта функция возвращает исполняемый прямо сейчас поток.

threading.main\_thread()Эта функция возвращает основной поток программы.

threading.enumerate() Эта функция возвращает список всех активных потоков.

threading.Timer() Эта функция модуля threading используется для создания нового потока и указания времени, через которое он должен запуститься. После запуска поток вызывает определенную функцию.

Модуль имеет встроенную функциональность для реализации блокировки, которая позволяет синхронизировать потоки. Блокировка необходима для контроля доступа к общим ресурсам для предотвращения повреждения или пропущенных данных.

Вы можете вызвать метод Lock(), чтобы применить блокировки, он возвращает новый объект блокировки. Затем вы можете вызвать метод захвата (блокировки) объекта блокировки, чтобы заставить потоки работать синхронно.

Необязательный параметр блокировки указывает, ожидает ли поток получения блокировки.

В случае, если блокировка установлена на ноль, поток немедленно возвращается с нулевым значением, если блокировка не может быть получена, и 1, если блокировка получена.

В случае, если для блокировки задано значение 1, поток блокируется и ожидает снятия блокировки.

Метод release() объекта блокировки используется для снятия блокировки, когда она больше не требуется, а **.acquire**() для получения блокировки.

### Socket

В Python для работы с сокетами используется модуль socket:

import socket

Прежде всего необходимо создать сокет:

sock = socket.socket()

Что бы связать наш сокет с хостом и портом нужно использовать метод bind, которому передается кортеж, первый элемент (или нулевой, если считать от нуля) которого — хост, а второй — порт:

sock.bind((host, port))

С помощью метода listen мы запустим для данного сокета режим прослушивания. Метод принимает один аргумент — максимальное количество подключений в очереди.

sock.listen(n)

Принять подключение можно с помощью метода accept, который возвращает кортеж с двумя элементами: новый сокет и адрес клиента. Именно этот сокет и будет использоваться для приема и посылке данных.

conn, addr = sock.accept()

Чтобы получить данные нужно воспользоваться методом recv, который в качестве аргумента принимает количество байт для чтения.

data = conn.recv(size)

Что бы отправить данные нужно воспользоваться методом send, который в качестве аргумента принимает количество байт для чтения.

data = conn.send(size)  
  
Закрыть соединение:

conn.close()

Что бы подключиться к серверу нужно использовать метод connect

sock.connect((host, port))

# Порядок выполнения работы

## Написание серверного приложения

Будет рассмотрен пример многопоточного серверного приложения на языке Python, позволяющего клиенту зашифровывать текст определенным шифром.

1. Произвести запуск виртуальной машины(сервер) и запустить терминал. Создать директорию server внутри директории /home/[имя пользователя]/ и внутри него создать файл server.py(рис.3).

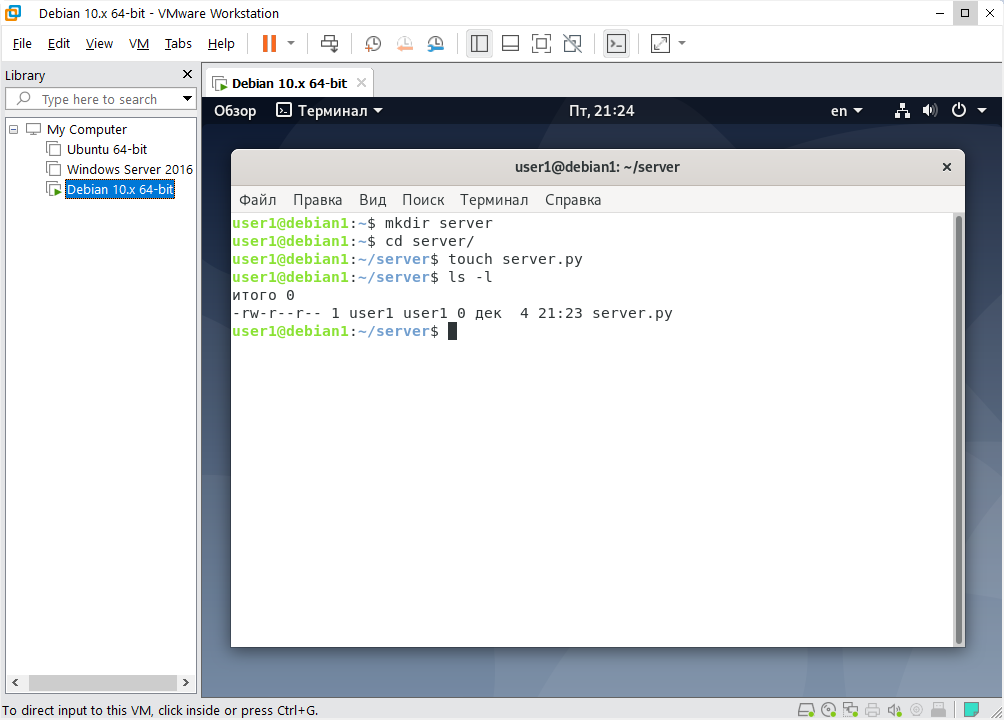


Рисунок – Создание файла server.py

1. Открыть файл server.py текстовым редактором для написания кода.

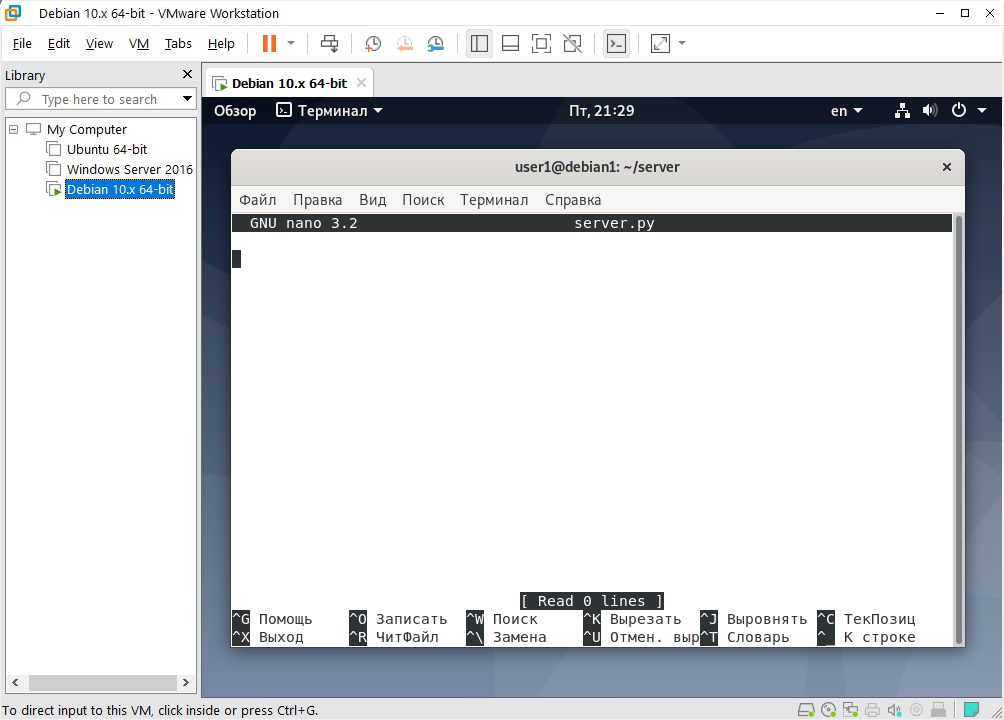


Рисунок – Редактирование файла server.py

1. Подключим необходимые Python библиотеки.

import socket

import treading

1. Определим основную функцию Main, в которой будет выполняться основной цикл серверного приложения. Параметрами данной функции являются address - IP-адрес сервера в текущей сети и arg\_port – порт через который будет создаваться TCP соединение.

def Main(arg\_address, arg\_port):

return 1

1. Определим внутри функции Main локальные переменные address и port и создадим сам сокет с помощью функции  
   socket(socket.AF\_INET, socket.SOCK\_STREAM), в которой AF\_INET означает, что подключение будет происходить по протоколу IPv4, а SOCK\_STREAM – что будет создан потоковый сокет, использующий TCP протокол при передаче данных. bind((host, port)) привязывает к созданному сокету адрес и порт, в рамках которых он будет работать.

host = arg\_address

port = arg\_port

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind((host, port))

print("socket binded to port", port)

1. Активируем прослушивание порта сокетом, используя метод listen(n), где n – максимальное число подключаемых одновременно сокетов-клиентов.

s.listen(10)

print("socket is listening")

1. Создадим цикл, в котором будет проводиться подключение сокетов и создание потоков для обработки запросов. При этом метод accept() производит соединение, запрашиваемое по привязанному адресу и порту; на выходе пара (conn, addr), где conn –объект сокета для работы по новому соединению, addr – адрес сокета, подключившийся к серверному сокету. Для обработки запросов от клиента создадим отдельный поток Thread(target = threadFunc, args = (conn,)), где threadFunc – функцию, которую будет выполнять ново созданный поток.

While True:

conn, addr = s.accept()

print('Connected to :', addr[0], ':', addr[1])

thread = threading.Thread(target = threadFunc, args = (conn,))

thread.start()

1. В конце функции Main необходимо закрыть серверный сокет методом close.

s.close()

1. Определим функцию threadFunc, исполняемую потоком, который создается при новом подключении сокета. Функция будет работать в цикле, принимая данные от клиентского сокета с помощью функции recv(n), где n – количество принимаемых байт. В случае окончания сеанса работы и получения команды --exit--, поток закрывает сокет(close()) и завершается. В обратном случае, текст шифруется с помощью функции cryptFunc(data) и отправляется обратно клиенту. Для этого используется вызов метода send(data).

def threadFunc(conn):

while True:

data = conn.recv(1024)

if not data:

continue

if data == '--exit--':

break

data = cryptFunc(data)

conn.send(data)

conn.close()

1. В качестве функции cryptFunc можно использовать любой алгоритм шифровки. Для примера, определим функцию следующим образом.

def cryptFunc(data):

data\_len = data = data[::-1]

return data

Весь код файла server.py:

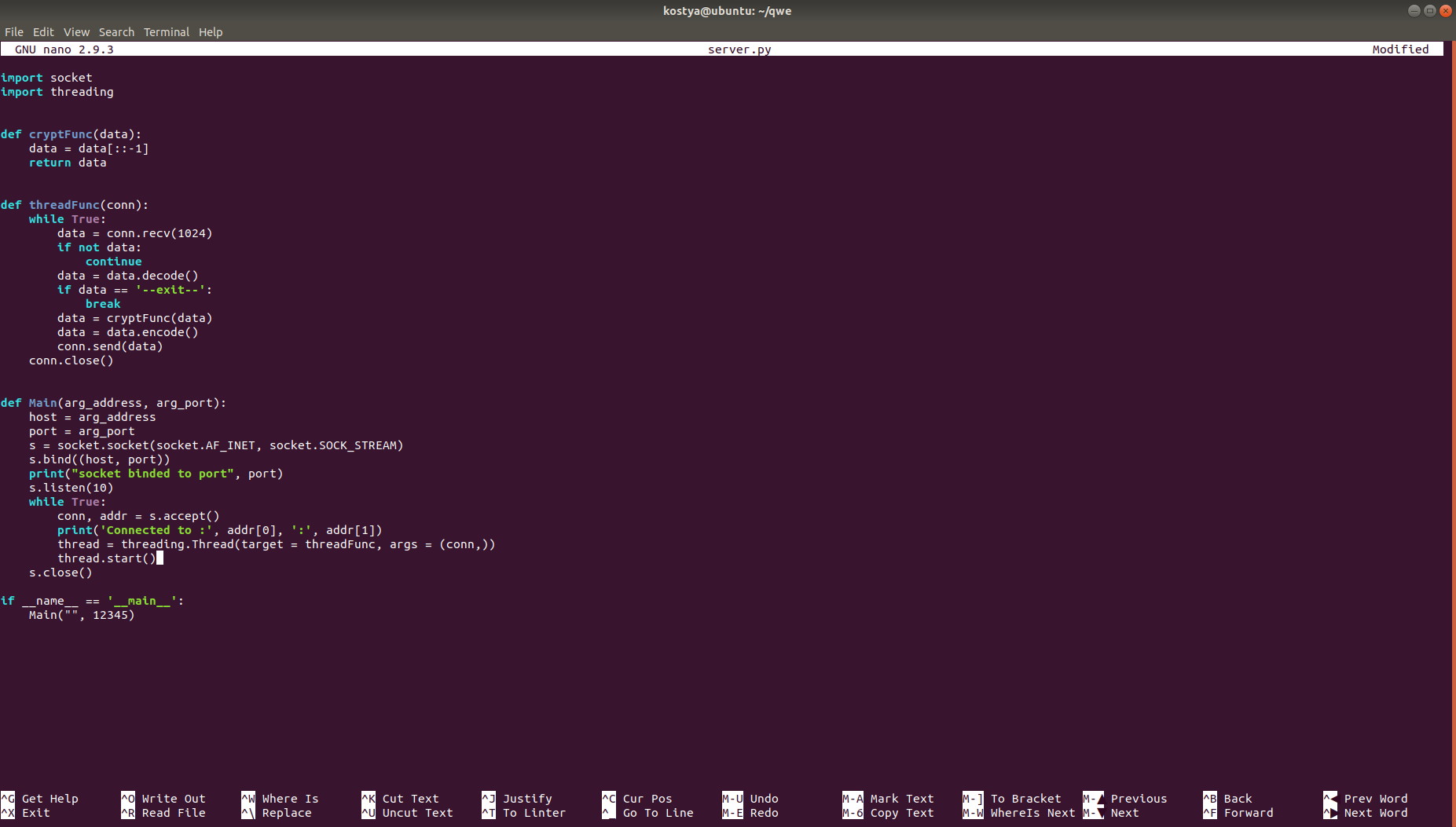


Рисунок 5 – Server.py

## Написание клиентского приложения

Будет рассмотрено написание клиентского приложения на языке Python, необходимого для работы с функционалом сервера.

1. Произвести запуск виртуальной машины(клиент) и запустить терминал. Создать директорию client внутри директории /home/[имя пользователя]/ и внутри него создать файл client.py(рис.3).

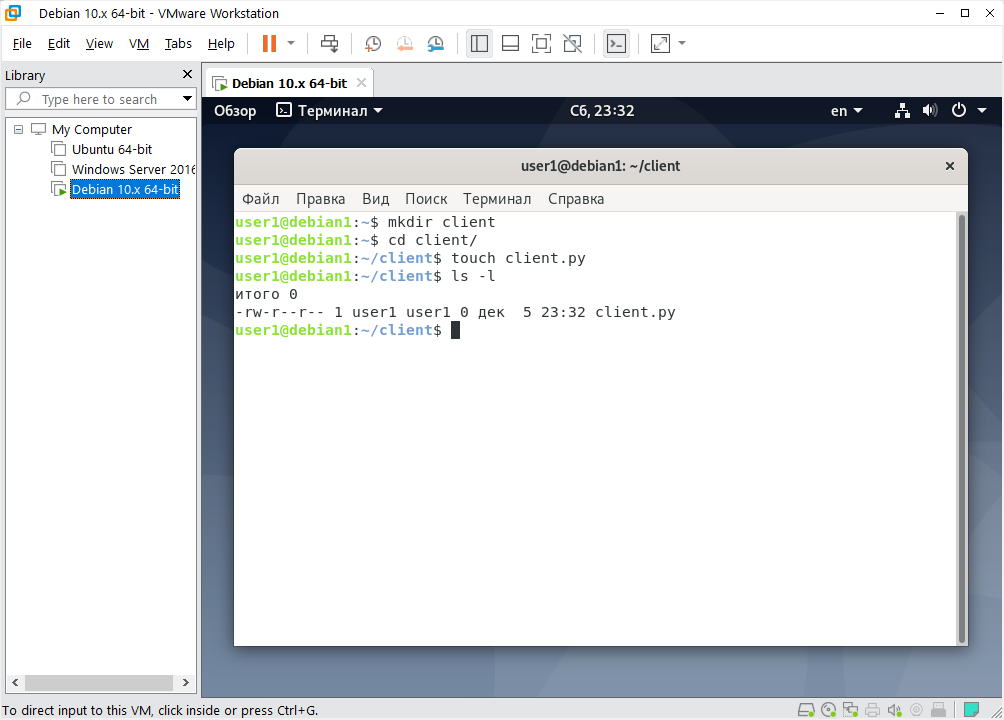


Рисунок 6 – Создание файла client.py

1. Открыть файл client.py текстовым редактором для написания кода.

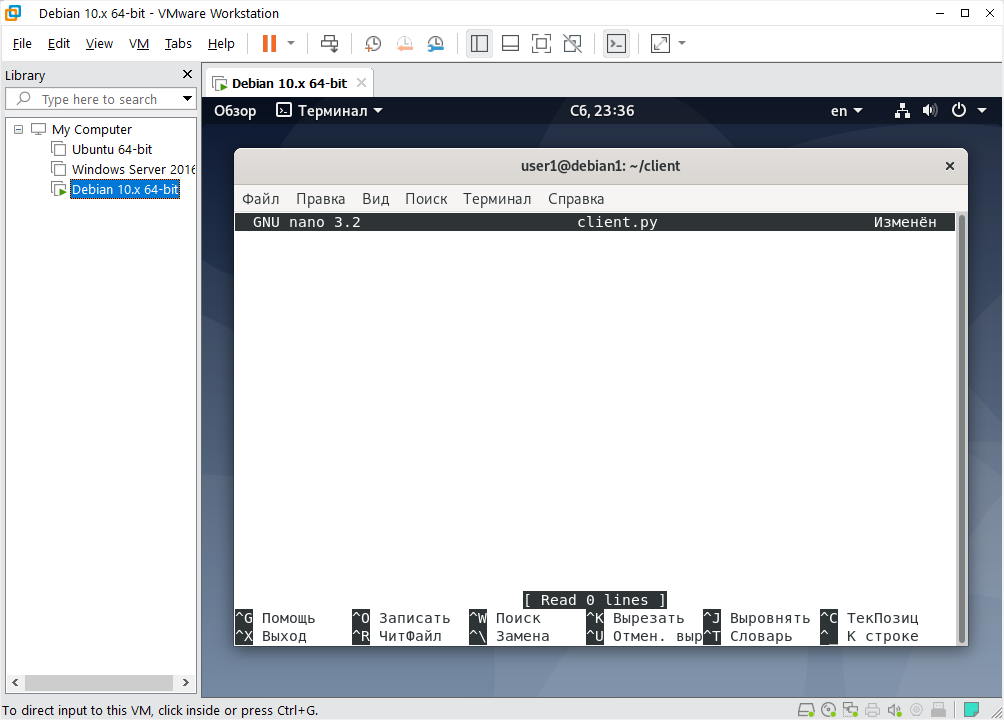


Рисунок 7 – Редактирование файла client.py

1. Подключим необходимые Python библиотеки.

import socket

1. Определим основную функцию Main, в которой будет выполняться основной цикл программы. Параметрами данной функции являются address - IP-адрес сервера в текущей сети и arg\_port – порт через который будет создаваться TCP соединение.

def Main(arg\_address, arg\_port):

return 1

1. Определим внутри функции Main локальные переменные address и port и создадим сам сокет с помощью функции  
   socket(socket.AF\_INET, socket.SOCK\_STREAM), в которой AF\_INET означает, что подключение будет происходить по протоколу IPv4, а SOCK\_STREAM – что будет создан потоковый сокет, использующий TCP протокол при передаче данных. Метод connect((host, port)) предпринимает попытку соединения с серверным сокетом на созданном порте и заданному адресу, в рамках которых он будет работать.

host = arg\_address

port = arg\_port

s = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.connect((host,port))

1. Определим само передаваемое сообщение в переменной message.

message = "Hello, Im your friend."

1. Создадим цикл, в котором будет проводиться передача сообщения и получение ответа. Метод send(data) позволит передавать данные data через соединение сокетов. Получить ответ от сервера можно с помощью recv(n), где n – количество получаемых байт. В конце итерации цикла необходимо ввести новые данные для шифрования, либо использовать служебную  
   команду «--exit--» для завершения сеанса связи.

while True:

s.send(message.encode())

if message == '--exit--':

break

data = s.recv(1024)

print('Received from the server :\t' + data.decode())

message = input()

1. В конце функции Main необходимо закрыть сокет методом close().

s.close()

Весь код файла client.py:

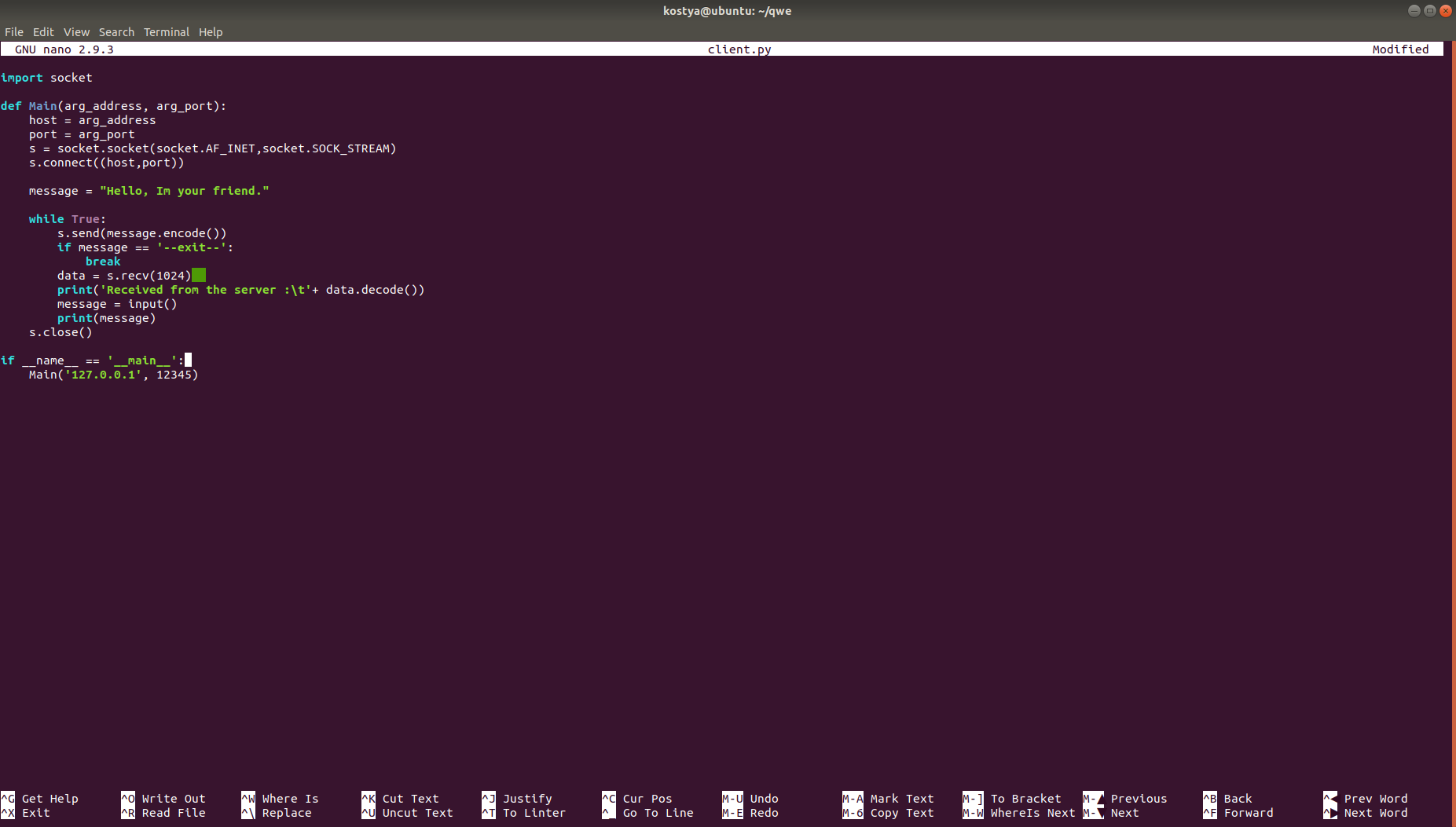


Рисунок 8 – client.py

# Самостоятельное задание

Необходимо написать клиент-серверное приложение (на базе приложения, написанного при выполнении л.р. по изучению сокетов), удовлетворяющее следующим требованиям:

1. Серверное приложение должно быть многопоточным, для обработки нескольких запросов одновременно.
2. Передача данным между клиентом и сервером проводится с помощью соединения сокетов.
3. Реализован способ корректного соединения и отсоединения клиента от сервера.
4. Реализовать передачу параметров через командную строку.

Пример выполнения подобного задания прилагается.

# Контрольные вопросы

1. В чем преимущества использования многопоточности?
2. Какие существуют недостатки данного подхода?
3. Из-за чего в Python существует проблема одновременной работы нескольких потоков одного процесса?