МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации

**

**ОТЧЁТ**

**по практической работе №3**

**«**Разработка сервиса сокращения ссылок**»**

**по дисциплине: «***Программирование***»**

Выполнил:Проверил:

Студент гр. «АБс-222», «АВТФ» *доцент кафедры ЗИ*

*Азява Д.А. Архипова А. Б.*

«\_\_\_» \_\_\_\_\_\_ 2023 г.«\_\_\_» \_\_\_\_\_\_ 2023 г.

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

(подпись) (подпись)

Новосибирск 2023

Цели и задачи работы: Требуется реализовать сервис для сокращения ссылок. Сервис должен принимать запросы по http протоколу.

Таблица 1 – Код программы

|  |  |
| --- | --- |
| index.html | <!DOCTYPE html>  <html>  <head>  <title>LinkShoter</title>  <style>  body{  background-image: url("https://sun9-18.userapi.com/impg/gaoA-FZD\_Qs7gL6bFPoMyGcgSx-FmzpEgoqxzA/8Kdcp9W5SRg.jpg?size=604x345&quality=95&sign=60542587d910c79ea8887ee9cf5aac13&type=album");  background-size: 100%;  }  h1{  text-align: center;  color: #fff;  font-size: 56px;  }  form{  text-align: center;  }  .url\_input{  color: #fff;  }  .url\_output{  color: #fff;  font-size: 28px;  }  .btm{  font-size: 14px;  }  </style>  </head>  <body>  <h1>LinkShoter</h1>  <form method="post" action="/">  <div >  <label  class="url\_input"  for="user\_input">Enter your URL: </label>  <input  type="text"  id="user\_input"  name="user\_input"  size="70"  required/>  <button class="btm" type="submit">Do it!</button>  </div>  {% if output\_link %}  <div class="url\_output" id="output\_link">{{output\_link}}</div>  {% endif %}  </form>  </body>  </html> |
| !index.html | <!DOCTYPE html>  <html>  <head>  <title>404</title>  <style>  body{  background-image: url("https://sun9-28.userapi.com/impg/c855628/v855628821/1af7fe/mDbggw9Im7Y.jpg?size=640x640&quality=96&sign=34bbcff56429bb736ab916ae9368b73f&type=album");  background-size: 100%;  }  h1{  text-align: center;  color: #000;  font-size: 56px;  }  </style>  </head>  <body>  <h1>Not Found</h1>  </body>  </html> |
| file.c | #include "hash\_table.h"  #include "file.h"  #include <stdio.h>  #include <string.h>  char\* decapitation(char\* input, char\* struc)  {  char\* output = strtok(input, "#");  if (strcmp(output, struc) == 0)  {  return strtok(NULL, "#");  }  else  {  return NULL;  }  }  char\* \_selection(char\* input, int start, int len)  {  int tmp = len - start;  char\* output = (char\*)calloc(tmp, sizeof(char));  output[tmp] = '\0';  for (int i = start, j = 0; i < len; i++, j++)  {  output[j] = input[i];  }  return output;  }  char\* read\_ht\_file(char\* path, HT\* table, char\* struc)  {  FILE\* mf = fopen(path, "r");  if (!mf)  {  printf("ERROR: The file does not exist\n");  return;  }  char mstr[100] = "Start";  char\* estr = "Start";  while (estr != NULL)  {  estr = fgets(mstr, sizeof(mstr), mf);  if((strstr(mstr, struc)) != NULL)  {  char\* data = \_selection(mstr, 9, strlen(mstr));  ht\_insert(table, struc, data);  return \_selection(mstr, 9, strlen(mstr));  }  }  return "NOT\_FOUND";  fclose(mf);  }  void write\_to\_file(char\* path, char\* struc, char\* data)  {  FILE\* mf = fopen(path, "a");  if (!mf)  {  printf("ERROR: Can't open the file\n");  return;  }  char\* result = strdup(struc);  char\* tmp = strdup(data);  fputs(result, mf);  fputs("#", mf);  fputs(tmp, mf);  fputs("\n", mf);  fclose(mf);  }  void clear\_file(char\* path)  {  FILE\* mf = fopen(path, "w");  if (!mf)  {  printf("ERROR: Can't open the file\n");  return;  }  fputs("", mf);  } |
| hash\_table.c | #include "hash\_table.h"  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #define CAPACITY 10 // Size of the Hash Table  List\* HEAD;  List\* TAIL;  //-----------------HASH FUNCK-------------------------------  unsigned long hash\_function(char\* str){  unsigned long i = 0;  for (int j = 0; str[j]; j++)  {  i += str[j];  }  return i % CAPACITY;  }  //-----------------CREATE HASH TABLE--------------------  ht\_node\* create\_item(char\* key, char\* value)  {  ht\_node\* item = (ht\_node\*)malloc(sizeof(ht\_node));  item->key = (char\*)malloc(strlen(key) + 1);  item->value = (char\*)malloc(strlen(value) + 1);  strcpy(item->key, key);  strcpy(item->value, value);  return item;  }  List\*\* create\_overflow(HT\* table)  {  List\*\* my\_list = (List\*\*)calloc(table->size, sizeof(List\*));  for (int i = 0; i < table->size; i++)  {  my\_list[i] = NULL;  }  return my\_list;  }  HT\* create\_table(int size)  {  HT\* table = (HT\*)malloc(sizeof(HT));  table->size = size;  table->count = 0;  table->items = (ht\_node\*\*)calloc(table->size, sizeof(ht\_node\*));  for (int i = 0; i < table->size; i++)  {  table->items[i] = NULL;  }  table->overflow = create\_overflow(table);  return table;  }  void free\_list(List\* list)  {  List\* temp = list;  while (list != NULL)  {  temp = list;  list = list->next;  free(temp->ht\_node->key);  free(temp->ht\_node->value);  free(temp->ht\_node);  free(temp);  }  }  void free\_overflow(HT\* table)  {  List\*\* my\_list = table->overflow;  for (int i = 0; i < table->size; i++)  {  free\_list(my\_list[i]);  }  free(my\_list);  }  void free\_item(ht\_node\* item)  {  // Frees an item  free(item->key);  free(item->value);  free(item);  }  void free\_table(HT\* table)  {  // Frees the table  for (int i = 0; i < table->size; i++)  {  ht\_node\* item = table->items[i];  if (item != NULL)  {  free\_item(item);  }  }  free\_overflow(table);  free(table->items);  free(table);  }  List\* list\_insert(List\* list, ht\_node\* item)  {  if (list == NULL)  {  list = (List\*)malloc(sizeof(List));  list->ht\_node = item;  list->next = NULL;  return list;  }  else if (list->next == NULL)  {  List\* tmp = (List\*)malloc(sizeof(List));  tmp->ht\_node = item;  tmp->next = NULL;  list->next = tmp;  return list;  }  else  {  while (list->next != NULL)  {  list = list->next;  }  List\* tmp = (List\*)malloc(sizeof(List));  tmp->ht\_node = item;  tmp->next = NULL;  list->next = tmp;  return list;  }  }  void handle\_collision(HT\* table, unsigned long index, ht\_node\* item)  {  List\* list = table->overflow[index];  if (list == NULL)  {  // We need to create the list  list = (List\*)malloc(sizeof(List));  list->ht\_node = item;  list->next = NULL;  table->overflow[index] = list;  return;  }  else {  // Insert to the list  table->overflow[index] = list\_insert(list, item);  return;  }  }  //----------------------PUSH TO HASH TABLE-------------------  void ht\_insert(HT\* table, char\* key, char\* value)  {  if (table == NULL)  {  return;  }  // Create the item  ht\_node\* item = create\_item(key, value);  int index = hash\_function(key);  ht\_node\* current\_item = table->items[index];  if (current\_item == NULL)  {  // Key does not exist.  if (table->count == table->size)  {  printf("Insert Error: Hash Table is full\n");  return;  }  // Insert directly  table->items[index] = item;  table->count++;  }  else  {  // Scenario 1: We only need to update value  if (strcmp(current\_item->key, key) == 0)  {  strcpy(current\_item->value, value);  //strcpy(table->items[index], current\_item->value);  return;  }  else  {  // Scenario 2: Collision  // We will handle case this a bit later  handle\_collision(table, index, item);  return;  }  }  }  char\* ht\_search(HT\* table, char\* key)  {  int index = hash\_function(key);  ht\_node\* item = table->items[index];  List\* list = table->overflow[index];  // Ensure that we move to a non NULL item  while (item != NULL)  {  if (strcmp(item->key, key) == 0)  {  return item->value;  }  if (list == NULL)  {  return NULL;  }  item = list->ht\_node;  list = list->next;  }  return NULL;  }  void print\_search(HT\* table, char\* key)  {  char\* val;  if ((val = ht\_search(table, key)) == NULL)  {  printf("Key:%s does not exist\n", key);  return;  }  else  {  printf("Key:%s, Value:%s\n", key, val);  }  }  void print\_table(HT\* table)  {  printf("\nHash Table\n-------------------\n");  for (int i = 0; i < table->size; i++)  {  if (table->items[i])  {  ht\_node\* tmp = table->items[i];  printf("Index:%d, Key:%s, Value:%s\n", i, tmp->key, tmp->value);  }  }  printf("-------------------\n\n");  }  void ht\_delete(HT\* table, char\* key)  {  // Deletes an item from the table  int index = hash\_function(key);  ht\_node\* item = table->items[index];  List\* head = table->overflow[index];  if (item == NULL)  {  // Does not exist. Return  return;  }  else  {  if (head == NULL && strcmp(item->key, key) == 0)  {  // No collision chain. Remove the item  // and set table index to NULL  table->items[index] = NULL;  free\_item(item);  table->count--;  return;  }  else if (head != NULL)  {  // Collision Chain exists  if (strcmp(item->key, key) == 0)  {  // Remove this item and set the head of the list as the new item  free\_item(item);  List\* list = head;  head = head->next;  list->next = NULL;  table->items[index] = create\_item(list->ht\_node->key, list->ht\_node->value);  free\_list(list);  table->overflow[index] = head;  return;  }  List\* curr = head;  List\* prev = NULL;  while (curr)  {  if (strcmp(curr->ht\_node->key, key) == 0)  {  if (prev == NULL)  {  // First element of the chain. Remove the chain  free\_list(head);  table->overflow[index] = NULL;  return;  }  else  {  // This is somewhere in the chain  prev->next = curr->next;  curr->next = NULL;  free\_list(curr);  table->overflow[index] = head;  return;  }  }  curr = curr->next;  prev = curr;  }  }  }  } |
| server.c | #include "hash\_table.c"  #include "file.c"  #include <stdio.h>  #include <string.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/types.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <netdb.h>  #include <fcntl.h>  #include <sys/mman.h>  #define msg\_success "SUCCESS"  #define msg\_error "ERROR:\_INCORRECT\_MESSAGE"  #define not\_found "ERROR:\_NOT\_FOUND"  #define \_PATH\_ "data.txt"  #define \_SIZE\_ 15  #define \_BUFF\_SIZE\_ 20 \* 1024  #define CLEAR\_BUFF memset(buff, '\0', sizeof(buff));  // функция обработки запросов  void dostuff (int sock, int shm);//, HT\* table);  // функция обработки ошибок  void error(const char \*msg) {  perror(msg);  exit(EXIT\_FAILURE);  }  // количество активных пользователей  int nclients = 0;  // печать количества активных пользователей  void printusers()  {  if (nclients) {  printf("%d user on-line\n", nclients);  }  else {  printf("No user on-line\n");  }  }  // Функция отделения  char\* \_decapitation(const char\* input, char\* word)  {  char\* output = strtok(input, "#");  if (strcmp(output, word) == 0)  {  return strtok(NULL, "#");  }  else  {  return NULL;  }  }  char\* selection(char\* input, int start, int len)  {  int tmp = len - start;  char\* output = (char\*)calloc(tmp, sizeof(char));  output[tmp] = '\0';  for (int i = start, j = 0; i < len; i++, j++)  {  output[j] = input[i];  }  return output;  }  int main(int argc, char \*argv[]) {  char buff[1024]; // Буфер для различных нужд  int sockfd, newsockfd; // дескрипторы сокетов  int portno; // номер порта  int pid; // id номер потока  socklen\_t clilen; // размер адреса клиента типа socklen\_t  struct sockaddr\_in serv\_addr, cli\_addr; // структура сокета сервера и клиента    // ошибка в случае если мы не указали порт  if (argc < 2) {  fprintf(stderr, "ERROR: no port provided\n");  exit(EXIT\_FAILURE);  }    // создание сокета  sockfd = socket(AF\_INET, SOCK\_STREAM, 0);  if (sockfd < 0) error("ERROR: opening socket");    // связывание сокета с локальным адресом  bzero((char\*) &serv\_addr, sizeof(serv\_addr));  portno = atoi(argv[1]);  serv\_addr.sin\_family = AF\_INET;  serv\_addr.sin\_addr.s\_addr = INADDR\_ANY; // сервер принимает подключения на все IP-адреса  serv\_addr.sin\_port = htons(portno);  if (bind(sockfd, (struct sockaddr \*) &serv\_addr, sizeof(serv\_addr)) < 0)  {  error("ERROR: on binding");  }  // ожидание подключений, размер очереди - 5  listen(sockfd, 5);  clilen = sizeof(cli\_addr);  // создание общей ячейки памяти для хранения количества активных пользователей  int shm = shm\_open("/shm", O\_CREAT | O\_RDWR, 0660);  if (shm == -1)  {  error("Shared memory open");  }  if (ftruncate(shm, sizeof(int)) == -1)  {  error("Shared memory truncate");  }  char\* addr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm, 0);  memcpy(addr, &nclients, sizeof(nclients));  HT\* table = create\_table(\_SIZE\_);  // извлекаем сообщение из очереди (цикл извлечения запросов на подключение)  while (1) {  newsockfd = accept(sockfd, (struct sockaddr \*) &cli\_addr, &clilen);  if (newsockfd < 0) error("ERROR: on accept");  memcpy(&nclients, addr, sizeof(nclients));  nclients++;  memcpy(addr, &nclients, sizeof(nclients));  // вывод сведений о клиенте  struct hostent \*hst;  hst = gethostbyaddr((char \*)&cli\_addr.sin\_addr, 4, AF\_INET);  printf("+%s [%s] new connect!\n",  (hst) ? hst->h\_name : "Unknown host",  (char\*)inet\_ntoa(cli\_addr.sin\_addr));  printusers();  pid = fork();  if (pid < 0) error("ERROR: on fork");  if (pid == 0) {  close(sockfd);  dostuff(newsockfd, shm);  exit(EXIT\_SUCCESS);  }  else close(newsockfd);  }  close(sockfd);  exit(EXIT\_SUCCESS);  }  void dostuff (int sock, int shm)  {  HT\* table = create\_table(\_SIZE\_);  int bytes\_recv; // размер принятого сообщения  char buff[\_BUFF\_SIZE\_];  char\* addr = mmap(0, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm, 0);  while (1)  {  // обработка первого параметра  bytes\_recv = read(sock, &buff[0], sizeof(buff));  char\* tmp = strdup(buff);  if (bytes\_recv < 0) error("ERROR reading from socket");  if (!strcmp(buff, "end")) break;  if(strstr(buff, "PUSH") != 0)  {  char\* key = \_decapitation(buff, "PUSH");  char\* data = selection(tmp, 14, strlen(tmp));  ht\_insert(table, key, data);  write\_to\_file(\_PATH\_, key, data);  write(sock, msg\_success, sizeof(msg\_success));  CLEAR\_BUFF  }  else if(strstr(buff, "GET") != 0)  {  char\* key = \_decapitation(buff, "GET");  char\* my\_msg = read\_ht\_file(\_PATH\_, table, key);  int len = strlen(my\_msg);  if(strcmp(my\_msg, "NOT\_FOUND"))  {  len--;  }  CLEAR\_BUFF  write(sock, my\_msg, sizeof(char)\*len);  }  else  {  write(sock, msg\_error, sizeof(msg\_error));  }  CLEAR\_BUFF  }  memcpy(&nclients, addr, sizeof(nclients));  nclients--; // уменьшаем счетчик активных клиентов  memcpy(addr, &nclients, sizeof(nclients));  close(sock);  printf("-disconnect\n");  printusers();  return;  } |
| service.py | from flask import Flask, render\_template, request, redirect  from urllib.parse import quote  import socket  app = Flask(\_\_name\_\_)  server\_address = ('127.0.0.1', 5656)  # Функция хэширования URL  def reduction(text):  # Добиваем строку, до кратности 8-ми  while ((len(text) % 8) != 0):  text += "#"    output = ['\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0']  # Определяем размер блоков  length = len(text) // 8  for i in range(0, 9):  res = 0  for j in range(0, (length \* i)):  res += ord(text[j]) \* ord(text[j]) \* j  res %= 1000  resulto = 0  # Суммируем цифры 3-х значного числа  while (res > 0):  resulto += res % 10  res //= 10    resulto += 60  output[i - 1] = chr(resulto)  return ''.join(output)  @app.route('/', methods=['GET', 'POST'])  def home():  new\_link = None  if request.method == 'POST':  real\_link = request.form['user\_input']  short\_link = reduction(real\_link)  # Сохраняем короткую ссылку в базе данных  with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:  try:  s.connect(server\_address)  s.sendall(f"PUSH#{short\_link}#{real\_link}\0'".encode())  print("Message sent successfully.")  data = s.recv(1024)  print(f"Answer from server: {data.decode()}")  s.sendall(f"end".encode())  except ConnectionRefusedError:  print("ERROR: Connection to the server failed.")  # Возвращаем короткую ссылку  new\_link = f"http://127.0.0.1:5000/{short\_link}"  return render\_template('index.html', output\_link=new\_link)  @app.route('/<short\_link>')  def redirect\_to\_real(short\_link):  if short\_link == 'favicon.ico':  return "Not Found"    # Получаем оригинальную ссылку из базы данных  with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:  try:  s.connect(server\_address)  s.sendall(f"GET#{short\_link}\0".encode())  data = s.recv(1024)  real\_link = data.decode()  print("catch: " + real\_link + "|end")  s.sendall(f"end".encode())  except ConnectionRefusedError:  print("Connection to the server failed.")  return "Internal Server Error"  # Открытие оригинальной ссылки  if real\_link != "NOT\_FOUND":  return redirect(real\_link)  else:  return render\_template('!index.html')  if \_\_name\_\_ == '\_\_main\_\_':  app.run(host='127.0.0.1', port=5000, debug=True) |

Перейдём по URL-ссылке «http://127.0.0.1:5000», появляется web-страница (Рисунок 1)

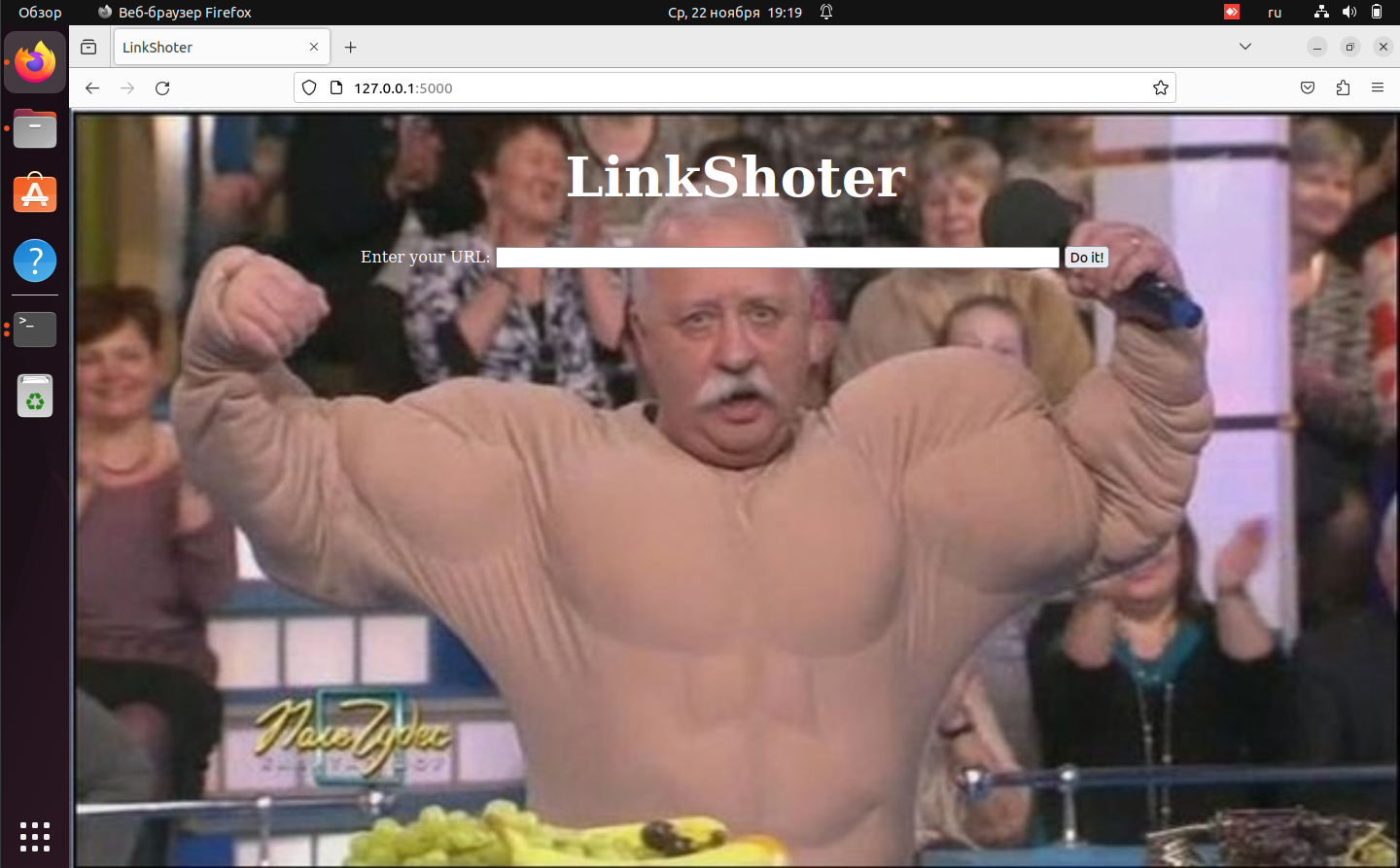


Рисунок 1 – web-страница

Введём ссылку, содержащую буквы латинского алфавита и кириллицы «https://мвд.рф/», в результате получаем сокращённую ссылку «http://127.0.0.1:5000/K<EJLCHL» (Рисунок 2)

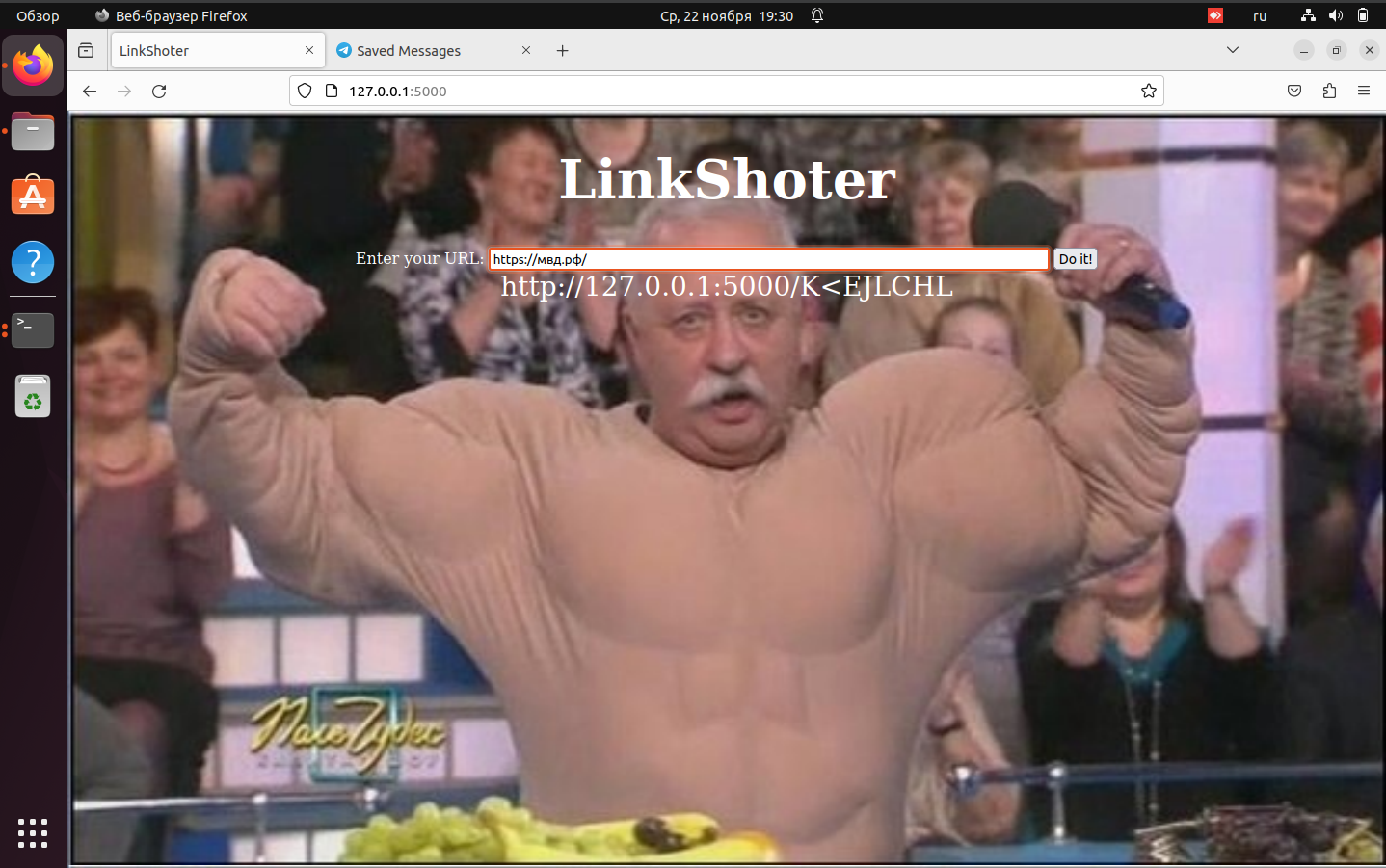


Рисунок 2 – Получение сокращённой ссылки

Введём в поисковую строку сокращённую ссылку «http://127.0.0.1:5000/K<EJLCHL», в результате наж запрос был перенаправлен на оригинальную ссылку «https://мвд.рф» (Рисунок 3)

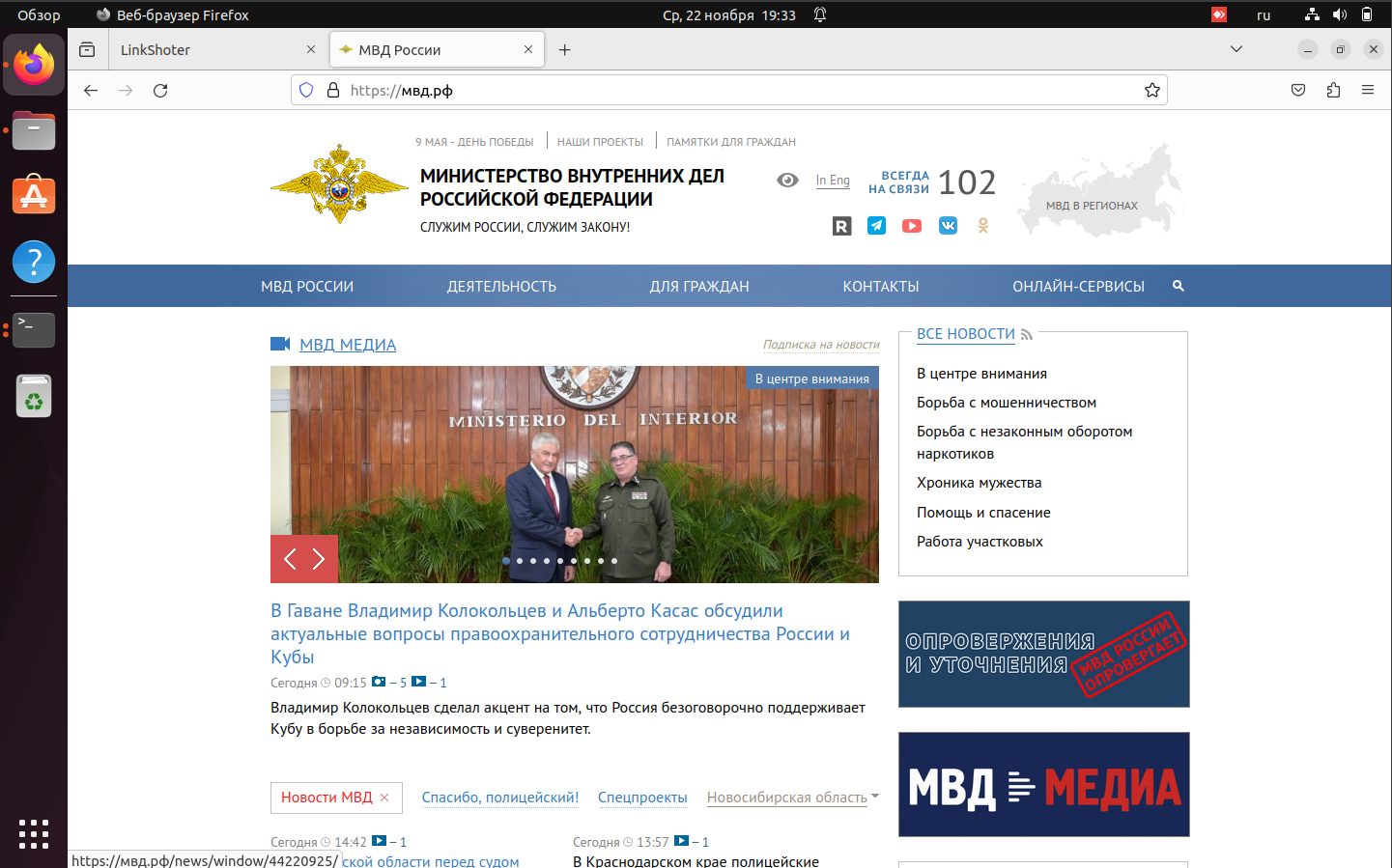


Рисунок 3 – Результат перехода по сокращённой ссылке

Введём несуществующую сокращенную ссылку «http://127.0.0.1:5000/AAAAAAAA», в результате показано сообщение о том, что данная ссылка не существует (Рисунок 4)

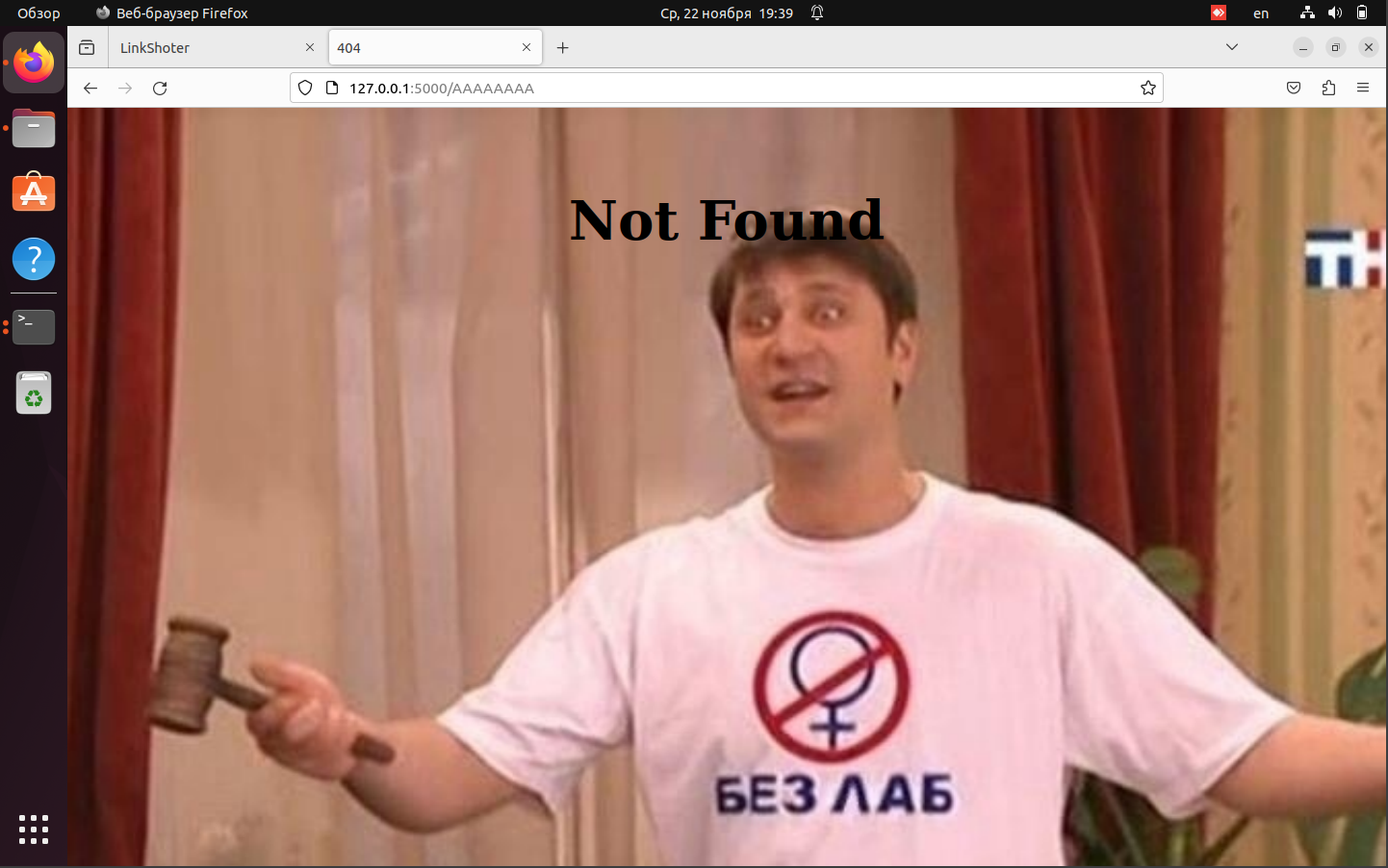


Рисунок 4 – Вывод ошибки 404

Вывод: в результате выполнения практической работы были получены навыки работы с HTML и написания сервисов обработки GET и POST запросов на ЯП Python.