Федеральное государственное автономное образовательное учреждение высшего профессионального образования Национальный исследовательский университет «Высшая школа экономики»

Московский институт электроники и математики Факультет прикладной математики и кибернетики

Кафедра «Компьютерная безопасность»

ОТЧЕТ

по дисциплине «Экзамен Современные технологии программирования и обработки информации»

Реализация асинхронного сервера с использование протокола FTP и библиотеки libevent

Выполнил: студент группы
СКБ-171 Зайцева А.А

МОСКВА 2021

Оглавление

Задание

Общее задание: Реализовать асинхронный параллельный сервер, реализующий выбранный задачей протокол. Порт приёма входящих соединений и число потоков задавать параметрами программы. Осуществлять штатный выход по сигналам прерывания/завершения процесса.

Задание для протокола FTP: Реализовать FTP-сервер. Достаточно реализовать только анонимный доступ (без аутентификации), пассивный режим (PASV, соединение для передачи данных устанавливает клиент), только команды, необходимые для перемещения по дереву каталогов, получению списка файлов и скачке файлов в поточном двоичном режиме. Тестировать корректность с браузерами, ещё поддерживающими FTP (не Chrome) или специализированными FTP-клиентами, например, WinSCP. По логам работы с клиентами можно выяснить минимальное необходимое количество команд для корректной работы.

Код реализации можно посмотреть здесь: Nasty09/Server_Ftp_Libevent

Протокол FTP

FTP может работать в активном или пассивном режиме, который определяет, как устанавливается соединение для передачи данных. В обоих случаях клиент создает управляющее соединение TCP из случайного порта N на командный порт FTP-сервера.

- В активном режиме клиент начинает прослушивать входящие соединения для передачи данных от сервера на порту M. Он отправляет команду FTP PORT M, чтобы сообщить серверу, какой порт он прослушивает. Затем сервер инициирует канал данных для клиента со своего порта.
- В ситуациях, когда клиент находится за брандмауэром и не может принимать входящие TCP-соединения, может использоваться пассивный режим. В этом режиме клиент использует управляющее соединение для отправки команды PASV на сервер, а затем получает от сервера IP-адрес и номер порта сервера, которые затем клиент использует для открытия соединения для передачи данных от порта клиента на IP-адрес сервера.

Сервер отвечает через управляющее соединение трехзначными кодами состояния в ASCII с дополнительным текстовым сообщением. Например, «200» (или «200 OK») означает, что последняя команда была успешной. Цифры представляют собой код ответа, а необязательный текст представляет понятное человеку объяснение или запрос. Текущая передача данных файла по соединению для передачи данных может быть прервана с помощью сообщения прерывания, отправленного по управляющему соединению.

Реализованные команды

PWD – возвращает текущий каталог хоста.

LIST – возвращает информацию о файле или каталоге, если он указан, иначе возвращается информация о текущем рабочем каталоге.

CWD – изменяет рабочую директорию

CDUP – изменяет родительский каталог

PORT – принимает адрес и порт, к которому должен подключиться сервер

RETR – получить копию файла

STOR – принять и сохранить данные в виде файла на сайте сервера

Ответы сервера

150 – статус файла в порядке; собирается открыть соединение для передачи данных

200 – действие было успешно завершено

220 – сервис готов для нового пользователя

226 – закрытие подключения для передачи данных. Запрошенное действие с файлом выполнено успешно

250 – запрошенное действие с файлом выполнено

257 – "PATHNAME" создано

450 – запрошенное действие с файлом не выполнено

501 – синтаксическая ошибка в параметрах или аргументах

Использованные функции библиотеки libevent

Папка event2 с заголовочными файлами для libevent:

- bufferevent.h:
  - *bufferevent_trigger* – запускает обратные вызовы данных bufferevent

```
void bufferevent_trigger(struct bufferevent *bufev, short iotype,
    int options);
```

  - *bufferevent_socket_new* – создает новый сокет bufferevent поверх существующего сокета

```
struct bufferevent *bufferevent_socket_new(struct event_base *base,
                                           evutil_socket_t fd, int options);
```

  - *bufferevent_socket_connect* – запускает попытку connect() с помощью bufferevent на основе сокетов

```
int bufferevent_socket_connect(struct bufferevent *,
                               const struct sockaddr *, int);
```

  - *bufferevent_set_timeouts* – устанавливает таймер на чтение или запись для bufferevent. Если чтение/запись отключены, или если операция чтения/записи bufferevent была приостановлена из-за отсутствия данных для записи, недостаточной пропускной способности и т.д., timeout не активен. timeout становится активным только тогда, когда мы действительно хотим читать или писать.

```
int bufferevent_set_timeouts(struct bufferevent *bufev,
const struct timeval *timeout_read, const struct timeval *timeout_write)
```

  - *bufferevent_write* – записывает данные в буфер bufferevent

```
int bufferevent_write(struct bufferevent *bufev,
    const void *data, size_t size);
```

  - *bufferevent_read* – читает данные из буфера bufferevent

```
size_t bufferevent_read(struct bufferevent *bufev,
                        void *data, size_t size);
```

  - *bufferevent_setcb* – изменяет обратные вызовы для события bufferevent

```
void bufferevent_setcb(struct bufferevent *bufev,
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void *cbarg);
```

  - *bufferevent_enable* – включает buffevent

```
int bufferevent_enable(struct bufferevent *bufev, short event);
```

  - *bufferevent_free* – освобождает память, связанную со структурой bufferevent

```
void bufferevent_free(struct bufferevent *bufev);
```

- util.h:
  - *evutil_inet_pton* – преобразует сетевой адрес IPv4 или IPv6 в его стандартной текстовой форме представления в его числовую двоичную форму
    ```
    int evutil_inet_pton(int af, const char *src, void *dst);
    ```
- event.h:
  - *event_base_dispatch* – создает цикл диспетчеризации событий. Он будет запускать базу событий до тех пор, пока не кончатся ожидающие или активные события, или пока не будут вызваны event_base_loopbreak() или event_base_loopexit()
    ```
    int event_base_dispatch(struct event_base *);
    ```
  - *event_base_free* – освобождает всю память, связанную с базой событий event_base, и освобождает base
    ```
    void event_base_free(struct event_base *);
    ```
  - *event_config_new* – создает объект конфигурации события. С помощью этой конфигурации можно менять поведение базы событий (event_base)
    ```
    struct event_config *event_config_new(void);
    ```
  - *event_config_set_flag* – устанавливает один или несколько флагов для настройки того, какие части возможной event_base будут инициализированы и как они будут работать
    ```
    int event_config_set_flag(struct event_config *cfg, int flag);
    ```
  - *event_base_new_with_config* – инициализирует новую базу событий с учетом указанной конфигурации
    ```
    struct event_base *event_base_new_with_config(const struct event_config *);
    ```
  - *event_config_free* – освобождает всю память, связанную с объектом конфигурации событий
    ```
    void event_config_free(struct event_config *cfg);
    ```
  - *event_new* – выделяет и назначает новую структуру событий, готовую к добавлению. Функция возвращает новое событие, которое можно использовать в будущих вызовах event_add() и event_del(). Аргументы fd и events определяют, какие условия вызовут событие; аргументы callback и callback_arg сообщают Libevent, что делать, когда событие становится активным
    ```
    struct event *event_new(struct event_base *, evutil_socket_t,
                            short, event_callback_fn, void *);
    ```
  - *event_add* – добавляет событие в набор ожидающих событий
    ```
    int event_add(struct event *ev, const struct timeval *timeout);
    ```

- listener.h:
  - *evconnlistener_new_bind* – выделяет новый объект evconnlistener для прослушивания входящих TCP-соединений по заданному адресу

```
struct evconnlistener *evconnlistener_new_bind(struct event_base *base,
    evconnlistener_cb cb, void *ptr, unsigned flags, int backlog,
    const struct sockaddr *sa, int socklen);
```

  - *evconnlistener_free* – отключает и освобождает evconnlistener

```
void evconnlistener_free(struct evconnlistener *lev);
```

Реализация

Сервер был реализован с помощью 11 файлов:

- main.cpp – создает список каналов и открывает первое событие для прослушивания порта
- Server.hpp – содержит все классы сервера: Task, Thread, TreadPool, Ftp_Factory, Ftp_Task, Ftp_List, Ftp_Port, Ftp_Retr, Ftp_Server_CMD, Ftp_Stor

- Thread.cpp
- ThreadPool.cpp
- Ftp_Factory.cpp
- Ftp_Task.cpp
- Ftp_List.cpp
- Ftp_Port.cpp
- Ftp_Retr.cpp
- Ftp_Server_CMD.cpp
- Ftp_Stor.cpp

Содержат описание функций/методов для одноименных классов

Описание классов

Task – базовый класс, описывающий событие. Содержит в себе 3 параметра:

```
class Task {
public:
    event_base *base = nullptr;
    int sock = 0;
    int thread_id = 0;
    virtual bool Init() = 0;
};
```

- *base* – указатель на базу событий;
- *sock* – дескриптор сокета, где произошло событие;
- *tread_id* – номер канала, в котором произошло событие.

Также есть виртуальная функция *Init()* для инициализации.

```
class Thread {
    int notify_send_fd = 0;
    event_base *base = nullptr;
    std::list<Task *> tasks;
    std::mutex tasks_mutex;
public:
    Thread() = default;
    ~Thread() = default;
    void Start();
    void Main();
    bool Setup();
    void Notify(evutil_socket_t fd, short which);
    void Activate();
    void AddTask(Task *task);
    int id = 0;
};
```

Thread – класс, описывающий работу канала.

Параметры:

- *notify_send_fd* – дескриптор канала записи для указания состояния канала;
- *base* – указатель на базу событий;
- *tasks* – список *task* канала;
- *tasks_mutex* – mutex канала;
- *Id* – номер канала.

Функции/методы:

- *Thread() / ~Thread()* – конструктор и деструктор;
- *Start()* – вызывает функцию Setup(), а затем, на этой основе, создает канал с помощью метода Main(). После выхода из Main() удаляет канал;
- *Main()* – создает цикл диспетчеризации событий и уничтожает базу событий, после завершения цикла;
- *Setup()* – создает канал pipe для чтения/записи, куда будет записываться состояние канала. Затем создается новая конфигурация события, которая не позволяет закрывать базу событий, и на ее основе задается *base*. Создается новое событие для чтения, которое добавляется в очередь;
- *Notify()* – выбирает первый элемент списка *tasks*, удаляет его и посылает на выполнение;
- *Activate()* – переводит канал в активное состояние;
- *AddTask()* – добавляет *task* в список *tasks*.

TreadPool – объединяет все каналы.

```
class ThreadPool {
    ThreadPool() = default;
    int threadCount = 0;
    int lastThread = -1;
    std::vector<Thread *> threads;
public:
    ~ThreadPool() = default;
    static ThreadPool *GetInstance() {
        static ThreadPool threadPool;
        return &threadPool;
    }
    void Init(int num);
    void Dispatch(Task *task);
};
```

Параметры:

- *threadCount* – количество каналов;
- *lastThread* – номер последнего использованного канала;
- *threads* – вектор, содержащий каналы.

Функции/методы:

- *ThreadPool() / ~ThreadPool()* – конструктор и деструктор;
- *GetInstance()* – создает элемент класса и возвращает ссылку на него;
- *Init()* – создает необходимое количество каналов и добавляет их в *threads*;
- *Dispatch* – при получении Task переходит на следующий по списку канал (или первый), добавляет Task и активирует канал.

```
class Ftp_Factory {
    Ftp_Factory() = default;
public:
    static Ftp_Factory *GetInstance();
    static Task *CreateTask();
};
```

Ftp_Factory – «завод» классов.

Функции/методы:

- *Ftp_Factory()* – конструктор;
- *GetInstance()* – создает элемент класса и возвращает ссылку на него;
- *CreateTask()* – создает Task, для которого регистрирует команды, используемые сервером, и объявляет классы для их выполнения.

```
class Ftp_Task : public Task {
public:
    std::string curDir = "/";
    std::string rootDir = "..";
    Ftp_Task *cmdTask = nullptr;
    std::string ip = "";
    int port = 0;
    virtual void Read(bufferevent *bev) {}
    virtual void Write(bufferevent *bev) {}
    virtual void Event(bufferevent *bev, short what) {}
    void SetCallback(bufferevent *bev);
    bool Init() override;
    virtual void Parse(const std::string &type, const std::string &msg) {}
    void ResponseCMD(const std::string &msg);
    void ConnectPORT();
    void Send(const std::string &data);
    void Send(const char *data, int len);
    void Close();
protected:
    static void EventCB(bufferevent *bev, short what, void *arg);
    static void ReadCB(bufferevent *bev, void *arg);
    static void WriteCB(bufferevent *bev, void *arg);
    bufferevent *cmdbev = nullptr;
    FILE *fp = nullptr;
};
```

Ftp_Task – наследуется от *Task*. Описывает поведение Task в рамках протокола FTP.
Параметры:

- *curDir* – нынешний каталог;
- *rootDir* – root каталог;
- *cmdTask* – указатель на Task протокола;
- *ip* – IP клиента;
- *port* – PORT клиента;
- *cmdbev* – указатель на буфер события;
- *fp* – указатель на файл.

Функции/методы:

- *Read()* – виртуальная функция, описание которой не нужно в Ftp_Task;
- *Write()* – виртуальная функция, описание которой не нужно в Ftp_Task;
- *Event()* – виртуальная функция, описание которой не нужно в Ftp_Task;
- *SetCallback()* – озменяет обратные вызовы для buffevent. Теперь при вызове read/write/event будут вызываться ReadCB/WriteCB/EventCB. Активирует buffevent для чтения/записи;
- *Init()* – переписанная функция класса *Task*;
- *Parse()* – виртуальная функция, описание которой не нужно в Ftp_Task;
- *ResponseCMD()* – посылает ответ на команду в буфер события;
- *ConnectPORT()* – создает сокет для буфера событий и соединяет порт клиента и сервера через него;
- *Send()* – записывает данные в буфер события;
- *Close()* – освобождает буфер события и закрывает файловый дескриптор;
- *EventCB()* – перенаправляет в функцию Event();
- *ReadCB()* – перенаправляет в функцию Read();
- *WriteCB()* – перенаправляет в функцию Write().

Ftp_List – наследуется от *Ftp*_Task. Дает ответ на команды, связанные с работой с файлами: LIST, PWD, CWD, CDUP.
Функции/методы:

- *getDirData()* – возвращает информацию о файле;

- *getFilePermissions()* – возвращает информацию о правах доступа к файлу;
- *getFileTime()* – возвращает время последнего изменения файла;
- *getDirectoryCount()* – считает количество директорий;
- *Parse()* – обрабатывает команды сервера и формирует ответ;
- *Write()* – завершает работу с файлом: выводит сообщение об успешном выполнении команды и закрывает файл и буфер событий;
- *Event()* – возвращает сообщение о успешном/провальном подключении к буферу событий.

```
class Ftp_List : public Ftp_Task {
    static std::string getDirData(const std::string &path);
    static std::string getFilePermissions(
            const std::filesystem::directory_entry &f);
    static std::string getFileTime(
            const std::filesystem::directory_entry &f);
    static int getDirectoryCount(const std::filesystem::path &fp);
public:
    void Parse(const std::string &type, const std::string &msg) override;
    void Write(bufferevent *bev) override;
    void Event(bufferevent *bev, short what) override;
};
```

```
class Ftp_Port : public Ftp_Task {
    void Parse(const std::string &type,
               const std::string &msg) override;
};
```

Ftp_Port – наследуется от *Ftp*_Task. Устанавливает связь между клиентом и сервером.

Функции/методы:

- *Parse()* – осуществляет обмен информации о порте и IP адресе между клиентом и сервером.

```
class Ftp_Retr : public Ftp_Task {
    FILE *fp = nullptr;
    char buf[1024] = {0};
public:
    void Parse(const std::string &type,
               const std::string &msg) override;
    void Write(bufferevent *bev) override;
    void Event(bufferevent *bev, short what) override;
};
```

Ftp_Retr – наследуется от *Ftp*_Task. Формирует ответ на команду RETR.

Параметры:

- *fp* – указатель на файл;
- *buf* – буфер.

Функции/методы:

- *Parse()* – открывает файл на побайтовое чтение, подключается к клиенту и устанавливает буфер события на запись;
- *Write()* – считывает информацию из файла в буфер buf и отправляет ее. Повторяет до тех пор, пока есть что считывать. После окончания закрывает файл и буфер событий;
- *Event()* – возвращает сообщение о успешном/провальном подключении к буферу событий.

```
class Ftp_Server_CMD : public Ftp_Task {
    std::map<std::string, Ftp_Task *> calls;
    std::map<Ftp_Task *, int> calls_del;
public:
    Ftp_Server_CMD() = default;
    ~Ftp_Server_CMD();
    bool Init() override;
    void Read(bufferevent *bev) override;
    void Event(bufferevent *bev, short what) override;
    void Reg(const std::string &cmd, Ftp_Task *call);
};
```

Ftp_Server_CMD – наследуется от *Ftp*_Task.
Управляет сервером.
Параметры:

- *calls* – список вызовов;
- *calls_del* – список вызовов на удаление.

Функции/методы:

- *Ftp_Server_CMD() / ~Ftp_Server_CMD()* – конструктор и деструктор;
- *Init()* – инициализирует сервер;
- *Read()* – считывает команду сервера и создает task для ее выполнения;
- *Event()* – возвращает сообщение о провальном событии;
- *Reg()* – регистрация нового вызова.

```
class Ftp_Stor : public Ftp_Task {
    FILE *fp = nullptr;
    char buf[1024] = {0};
public:
    void Parse(const std::string &type,
               const std::string &msg) override;
    void Read(bufferevent *bev) override;
    void Event(bufferevent *bev, short what) override;
};
```

Ftp_Stor – наследуется от *Ftp*_Task.
Формирует ответ на команду STOR.
Парметры:

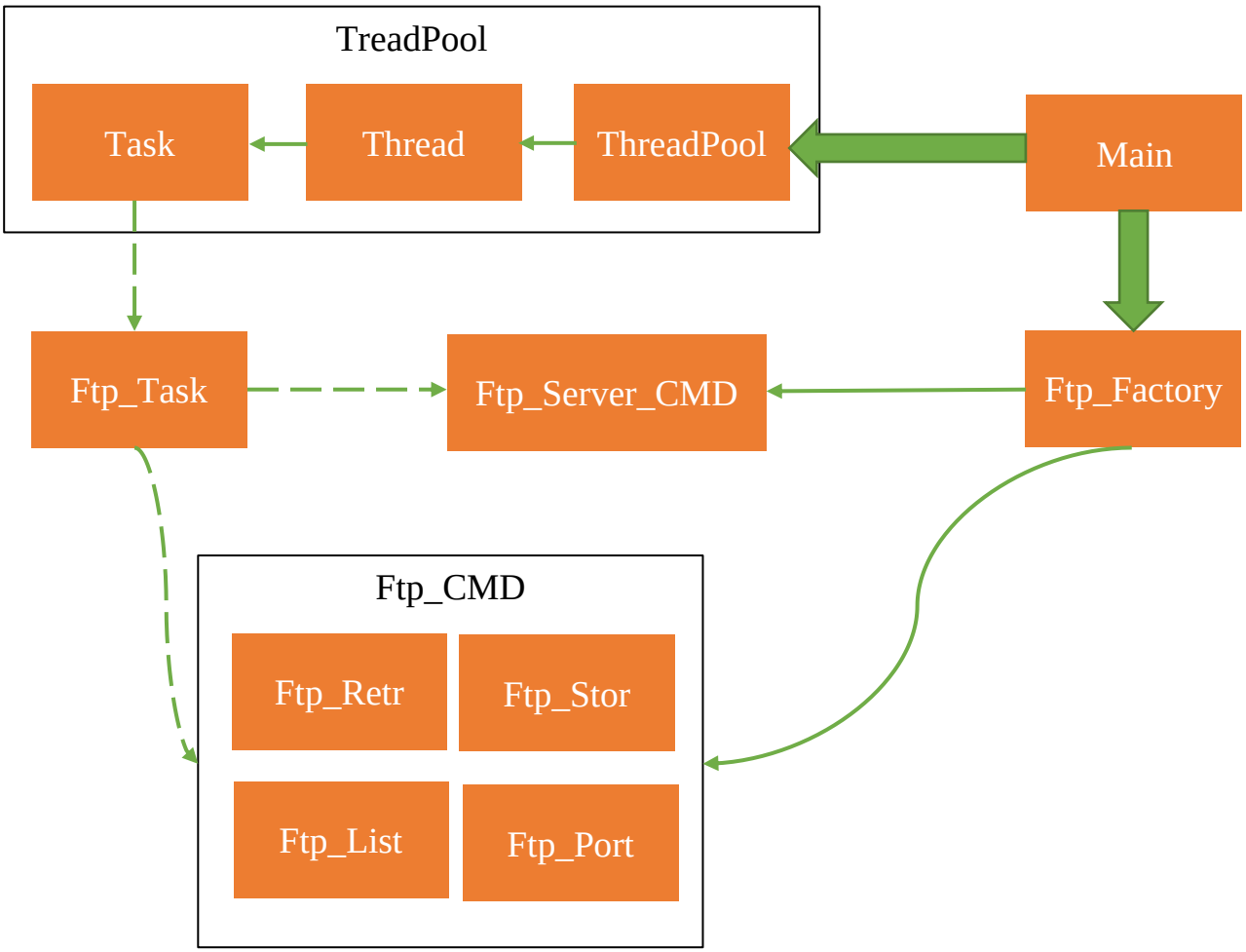
- *fp* – указатель на файл;
- *buf* – буфер.

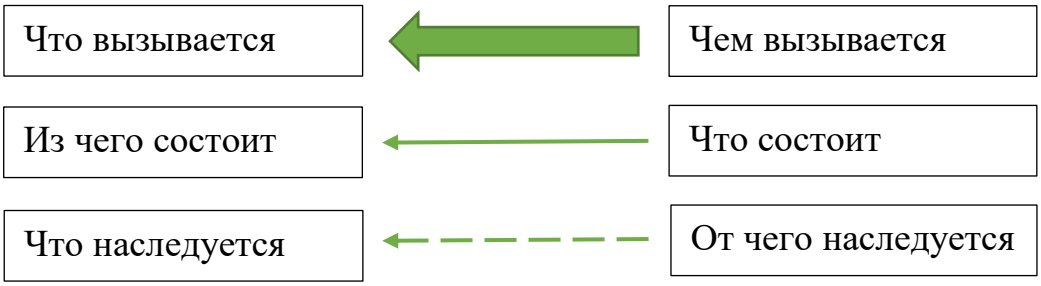
Функции/методы:

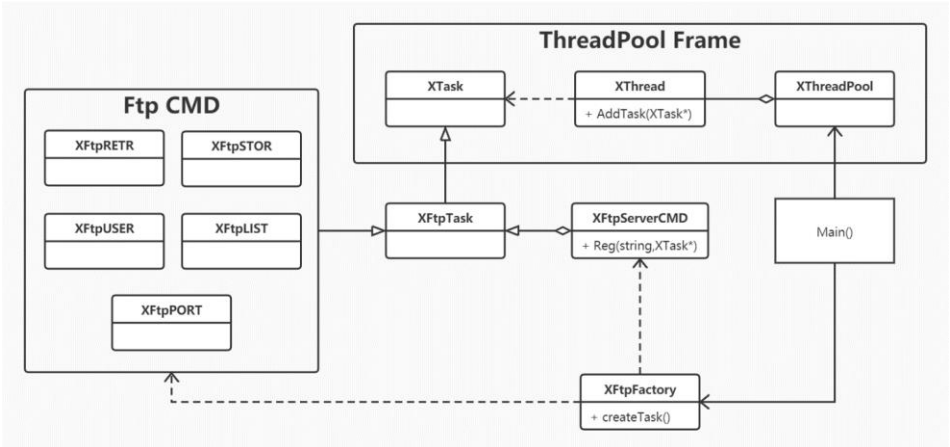
- *Parse()* – открывает файл на побайтовую запись, подключается к клиенту и устанавливает буфер события на чтение;
- *Read()* – считывает информацию из буфера событий в буфер и записывает в файл. Повторяет до тех пор, пока есть что считывать;
- *Event()* – возвращает сообщение о успешном/провальном подключении к буферу событий.

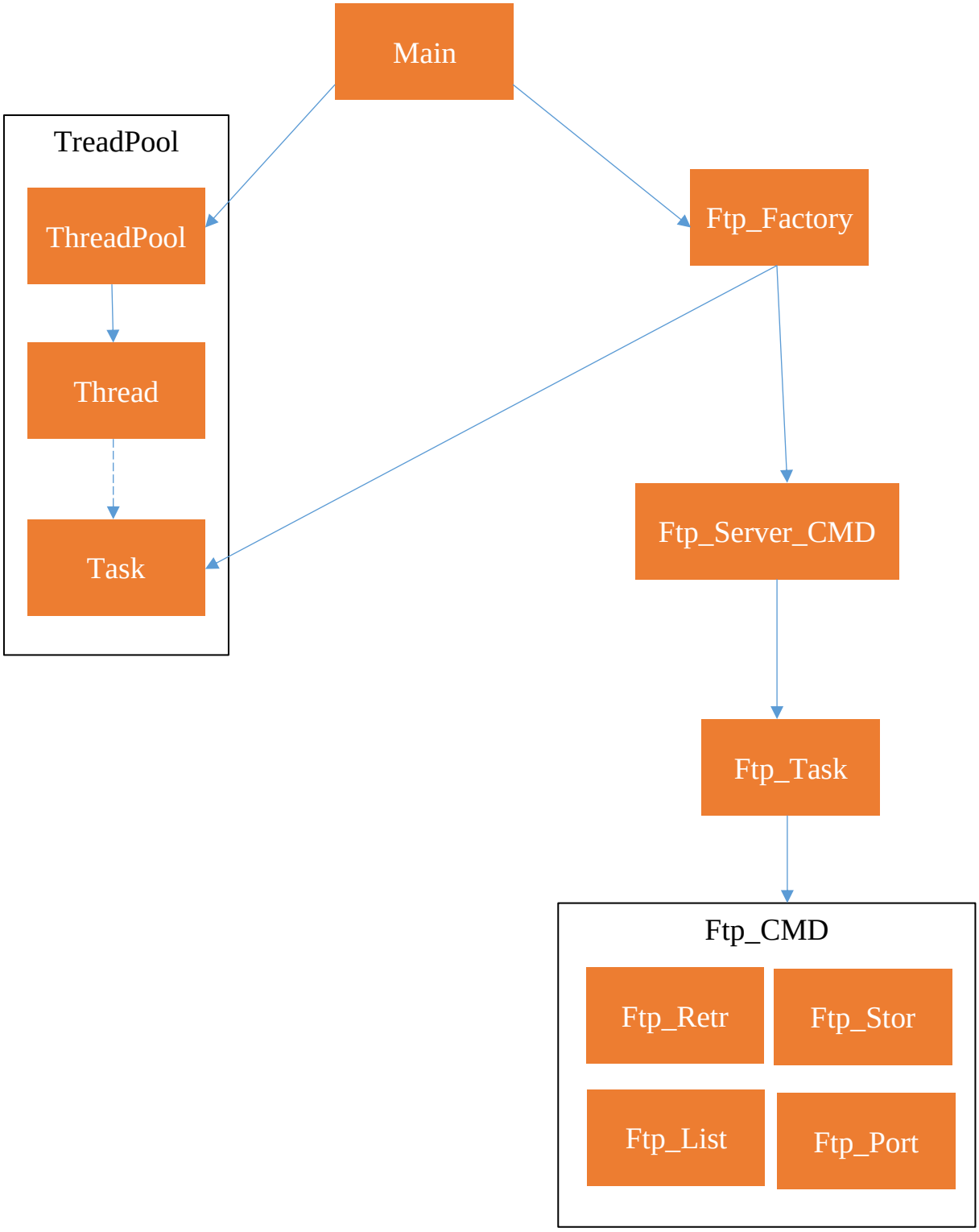
Описание алгоритма

main() порождает объект класса ThreadPool. Затем ThreadPool порождает N-ое количество объектов класса Thread и главное событие.
После того, как приходит запрос на сервер, событие переходит в режим прослушивания и создает объект класса Ftp_Factory и, с помощью него, объект класса Task в рамках i-ой Thread.
После прихода команды на сервер инициализируется объект класса Ftp_Server_CMD, который порождает объект класса Ftp_Task, который передается одному из классов обработки команд.


TreadPool
Task
Thread
ThreadPool
Main
Ftp_Task
Ftp_Server_CMD
Ftp_Factory
Ftp_CMD
Ftp_Retr
Ftp_Stor
Ftp_List
Ftp_Port

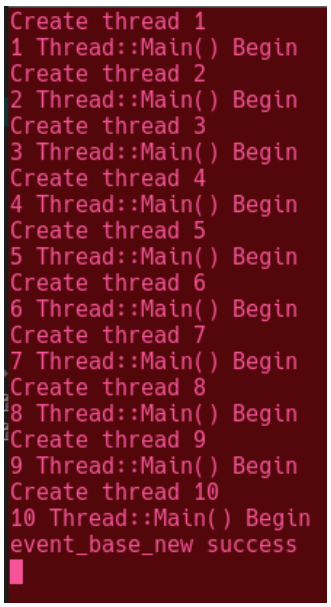

Что вызывается
Чем вызывается
Из чего состоит
Что состоит
Что наследуется
От чего наследуется


ThreadPool Frame
XTask
XThread
+ AddTask(XTask*)
XThreadPool
Ftp CMD
XFtpRETR
XFtpSTOR
XFtpUSER
XFtpLIST
XFtpPORT
XFtpTask
XFtpServerCMD
+ Reg(string,XTask*)
Main()
XFtpFactory
+ createTask()

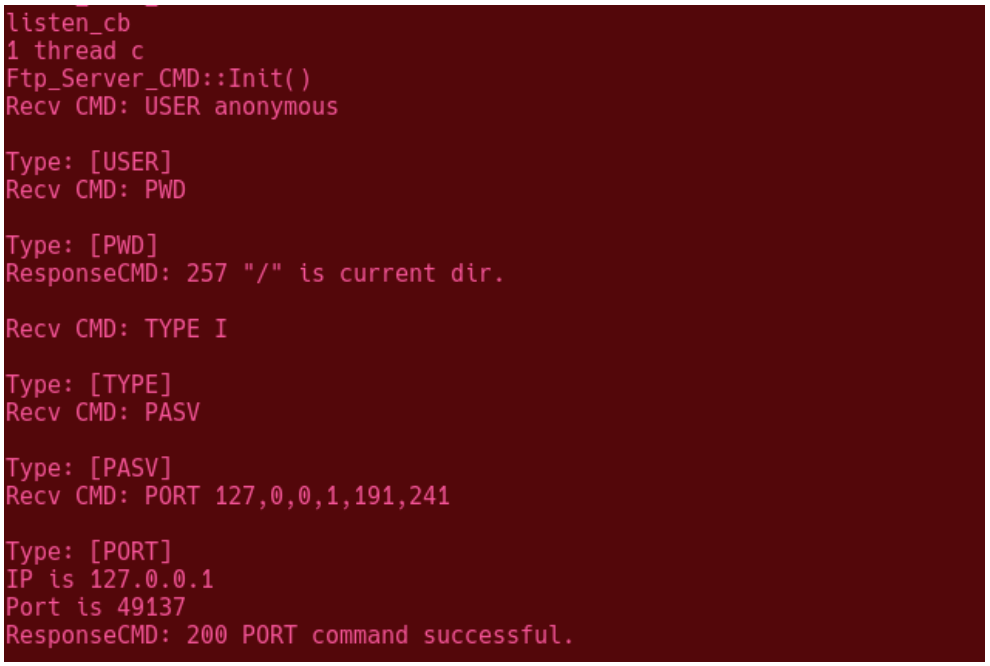

Main
TreadPool
ThreadPool
Thread
Task
Ftp_Factory
Ftp_Server_CMD
Ftp_Task
Ftp_CMD
Ftp_Retr
Ftp_Stor
Ftp_List
Ftp_Port

Тестирование

Тестирование сервера проводилось с помощью программы FileZilla – специализированная программа для тестирования соединений с использованием протокола FTP.
После запуска создается N-ое (в примере 10) количество каналов и первичная база событий:



После этого можно подключиться к серверу с помощью FileZilla:



Было начато прослушивание – listen_cb
Был активирован 1 канал – 1 thread c
Инициализирован FTP сервер – Ftp_Server_CMD::Init()

Сервер получает команду USER, но режим анониманый – Recv CMD: USER anonymous
Сервер получил команду PWD и послал в ответ нынешнюю директорию – ResponseCMD: 257 “/” is current dir.
Сервер настроен на активный режим, поэтому игнорирует команду PASV и принимает команду PORT
В ответ сервер посылает информацию об успешном выполнении команды – ReasponseCMD: 200 PORT command successful

Клиент запросил информацию о файле с помощью команды LIST
Сервер прислал уведомление о получении команды, подключился к каналу и передал информацию

```
Recv CMD: LIST

Type: [LIST]
../WorldOfPlanets/build-untitled-Clang-Debug "/home/user/Projects/Palas/build-Asinc_Server-Clang-Release/../WorldOfPl
anets/build-untitled-Clang-Debug"
ResponseCMD: 150 Here comes the directory listing.

BEV_EVENT_CONNECTED
ResponseCMD: 226 Transfer complete
```

Команда CWD

```
Recv CMD: CWD /WorldOfPlanets

Type: [CWD]
ResponseCMD: 250 Directory success changed.
```

Команда RETR для файла Move_Exmp.txt

```
Recv CMD: RETR Move_Exmp

Type: [RETR]
ResponseCMD: 150 File OK

BEV_EVENT_CONNECTED
ResponseCMD: 226 Transfer complete
```

Команда STOR для файла Move_EXMP_2.0.txt

```
Recv CMD: STOR Move_EXMP_2.0.txt

Type: [STOR]
ResponseCMD: 150 File OK

ResponseCMD: 226 Transfer complete
```

Список литературы

1. Библиотека libevent – http://www.wangafu.net/~nickm/libevent-book/
2. Описание протокола FTP – https://tools.ietf.org/html/rfc959
3. Информация о C++ – C++ reference - cppreference.com