Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Лабораторна робота №4

Мережеве програмування в середовищі Unix

Тема: TCP клієнт-сервер з мультиплексуванням введення-виведення

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірив:
Сімоненко А.В.

Київ 2025

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Розробити однопотоковий сервер з використанням мультиплексування введення-виведення для одночасної роботи з кількома клієнтами, застосовуючи системні виклики POSIX select() або poll().

2 ЗАВДАННЯ

Розробити однопотоковий сервер, який виконує наступне:

1. Сервер підтримує аргументи командного рядка, визначені в лабораторній роботі No3. Також сервер підтримує аргумент командного рядка, який визначає максимальну кількість клієнтів, з якими сервер може одночасно працювати. Сервер не приймає нові TCP з’єднання після досягнення цього значення.
2. Сервер працює з клієнтами відповідно до користувальницького протоколу, визначеного в лабораторній роботі No3.
3. Сервер дозволяє одночасно працювати з кількома клієнтами за допомогою мультиплексування введення-виведення. Сервер послуговується системними викликами select() або poll() для мультиплексування введення-виведення.
4. Кількість даних, які сервер зчитує або відправляє одному клієнту під час виконання введення-виведення з ним, треба обмежити. Ця кількість задається в коді сервера константою, яка може мати значення 1 байт та більше. Тобто, якщо сервер отримав інформацію від ядра про можливість виконати введення або виведення для якогось дескриптора файлу сокета, тоді серверу дозволено відправити або отримати даних розміром не більше вказаної константи. Це обмеження дає змогу майже порівну розподіляти час роботи сервера для кожного клієнта, який потребує комунікації. Також невеликі значення цієї константи дозволяють імітувати проблеми з мережею та частково імітувати різну поведінку клієнтів. Сервер не має завершувати своє виконання у випадку виникнення несистемної помилки. Рекомендації для сервера такі самі, які були дані в лабораторній роботі No3.

3 ВИКОНАННЯ

3.1 Виконання

Напишемо скрипт для тестування роботи сервера.

```
import subprocess
import pathlib
import os
import time

SCRIPT_DIR                                                    =
pathlib.Path(os.path.dirname(os.path.abspath(__file__)))
BUILD_DIR = SCRIPT_DIR / 'build'
CLIENT_EXECUTABLE = BUILD_DIR / 'client.o'
SERVER_EXECUTABLE = BUILD_DIR / 'multiplex_server.o'
ADDRESS  = '0.0.0.0'
PORT = '55001'
MAX_FILE_SIZE = '1000000000'
MAX_CLIENTS = '100'

BOOKS_DIR = pathlib.Path('/home/sideshowbobgot/university/C')

server  =  subprocess.Popen([SERVER_EXECUTABLE,  ADDRESS,  PORT,
BOOKS_DIR, MAX_CLIENTS])

clients: list[subprocess.Popen[bytes]] = []
for dirpath, dirnames, filenames in os.walk(BOOKS_DIR):
    for file in filenames:
        if file.endswith('.pdf'):
                clients.append(subprocess.Popen([CLIENT_EXECUTABLE,
ADDRESS, PORT, file, MAX_FILE_SIZE]))

for client in clients:
    client.wait()

time.sleep(1)
```

```
print('[CLIENTS FINISHED]')

server.wait()
```

Цей скрипт створює систему, де для кожного PDF-файлу в вказаній директорії запускається окремий клієнтський процес, який взаємодіє з центральним сервером.Скрипт імпортує необхідні бібліотеки для роботи з процесами, файловою системою та часом. Визначаються важливі шляхи та параметри: SCRIPT_DIR - директорія, де знаходиться сам скрипт; BUILD_DIR - директорія з скомпільованими файлами; CLIENT_EXECUTABLE - шлях до виконуваного файлу клієнта; SERVER_EXECUTABLE - шлях до виконуваного файлу сервера; ADDRESS - IP-адреса ('0.0.0.0' означає прослуховування на всіх інтерфейсах); PORT - порт для з'єднання ('55002'); MAX_FILE_SIZE - максимальний розмір файлу ('1000000000' байтів або приблизно 1 ГБ); BOOKS_DIR - директорія з PDF-файлами. Спочатку запускається сервер як окремий процес за допомогою subprocess.Popen. Створюється порожній список clients для зберігання клієнтських процесів. Скрипт рекурсивно проходить через усі піддиректорії BOOKS_DIR за допомогою os.walk(). Для кожного файлу з розширенням '.pdf', який знаходиться в директорії запускається новий клієнтський процес, кожному клієнту передаються параметри: адреса сервера, порт, ім'я файлу та максимальний розмір файлу, процес додається до списку clients. Після створення всіх клієнтів, скрипт чекає завершення кожного клієнтського процесу за допомогою client.wait(). Після завершення всіх клієнтів скрипт виводить повідомлення '[CLIENTS FINISHED]'. Нарешті, скрипт чекає завершення серверного процесу з server.wait(). Цей підхід дозволяє обробляти багато файлів паралельно, оскільки для кожного PDF-файлу створюється окремий процес, який може взаємодіяти з сервером незалежно від інших.

На рисунку 3.1 зображено роботу сервера, до якого звертаються клієнти. Як бачимо з логів, клієнти оброблюються конкурентно.

```
[client_fd: 44] [ClientStateTag_SEND_CHUNK]
[client_fd: 46] [ClientStateTag_SEND_CHUNK]
[client_fd: 50] [ClientStateTag_SEND_CHUNK]
[client_fd: 52] [ClientStateTag_SEND_CHUNK]
```

```
[client_fd: 54] [ClientStateTag_SEND_CHUNK]
[client_fd: 8] [ClientStateTag_SEND_CHUNK]
[client_fd: 10] [ClientStateTag_SEND_CHUNK]
[client_fd: 14] [ClientStateTag_SEND_CHUNK]
[client_fd: 16] [ClientStateTag_SEND_CHUNK]
[client_fd: 22] [ClientStateTag_SEND_CHUNK]
[client_fd: 24] [ClientStateTag_SEND_CHUNK]
[client_fd: 28] [ClientStateTag_SEND_CHUNK]
[client_fd: 30] [ClientStateTag_SEND_CHUNK]
[client_fd: 44] [ClientStateTag_SEND_CHUNK]
[client_fd: 46] [ClientStateTag_SEND_CHUNK]
[client_fd: 50] [ClientStateTag_SEND_CHUNK]
[client_fd: 52] [ClientStateTag_SEND_CHUNK]
[client_fd: 54] [ClientStateTag_SEND_CHUNK]
[client_fd: 8] [ClientStateTag_SEND_CHUNK]
[client_fd: 10] [ClientStateTag_SEND_CHUNK]
[client_fd: 14] [ClientStateTag_SEND_CHUNK]
[client_fd: 16] [ClientStateTag_SEND_CHUNK]
[client_fd: 22] [ClientStateTag_SEND_CHUNK]
[client_fd: 24] [ClientStateTag_SEND_CHUNK]
[client_fd: 28] [ClientStateTag_SEND_CHUNK]
[client_fd: 30] [ClientStateTag_SEND_CHUNK]
[client_fd: 44] [ClientStateTag_SEND_CHUNK]
[client_fd: 46] [ClientStateTag_SEND_CHUNK]
[client_fd: 50] [ClientStateTag_SEND_CHUNK]
[client_fd: 52] [ClientStateTag_SEND_CHUNK]
[client_fd: 54] [ClientStateTag_SEND_CHUNK]
[client_fd: 8] [ClientStateTag_SEND_CHUNK]
[client_fd: 10] [ClientStateTag_SEND_CHUNK]
[client_fd: 14] [ClientStateTag_SEND_CHUNK]
[client_fd: 16] [ClientStateTag_SEND_CHUNK]
[client_fd: 22] [ClientStateTag_SEND_CHUNK]
[client_fd: 24] [ClientStateTag_SEND_CHUNK]
[client_fd: 28] [ClientStateTag_SEND_CHUNK]
[client_fd: 30] [ClientStateTag_SEND_CHUNK]
[client_fd: 44] [ClientStateTag_SEND_CHUNK]
[client_fd: 46] [ClientStateTag_SEND_CHUNK]
```

```
[client_fd: 50] [ClientStateTag_SEND_CHUNK]
[client_fd: 52] [ClientStateTag_SEND_CHUNK]
[client_fd: 54] [ClientStateTag_SEND_CHUNK]
```

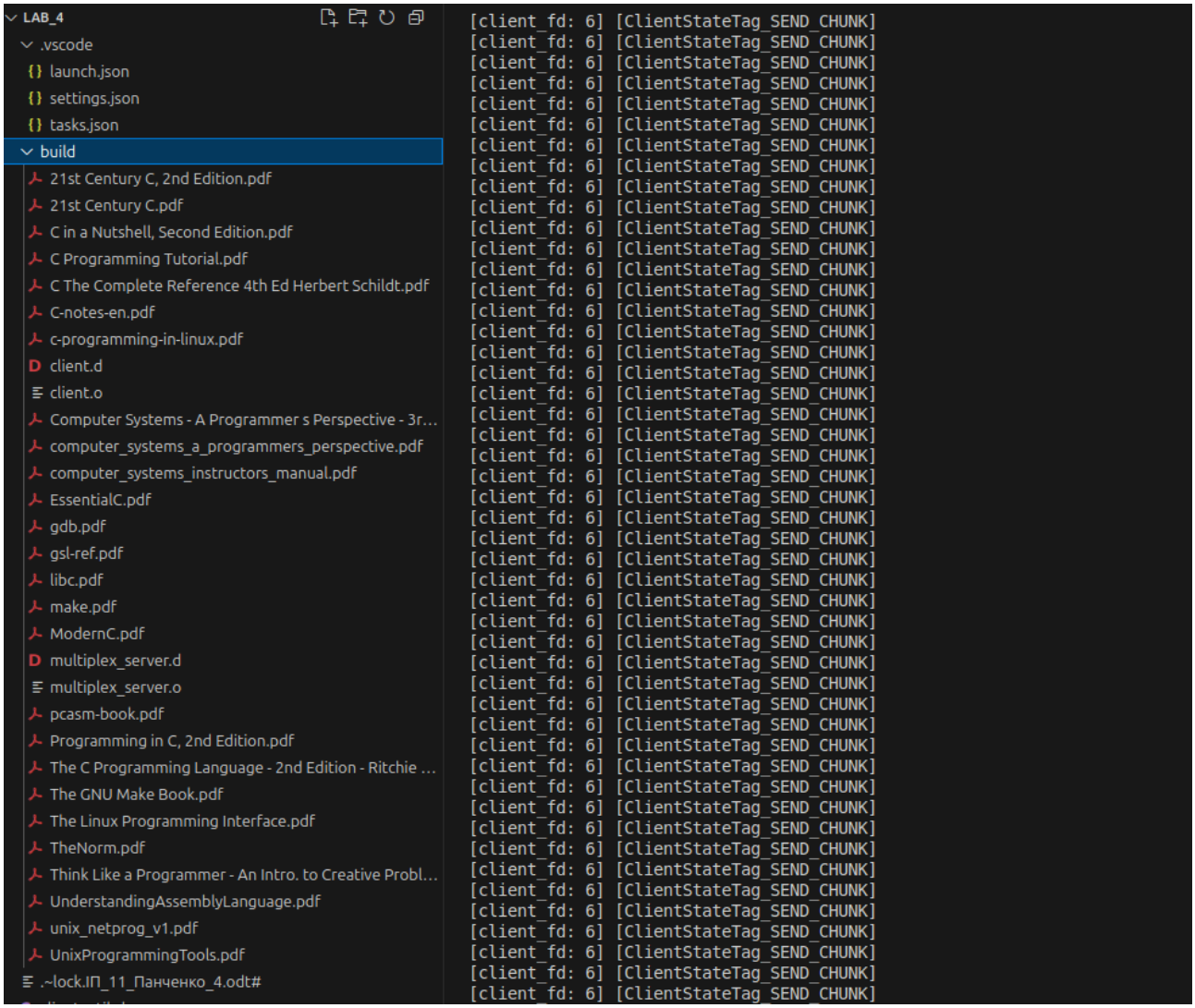

Рисунок 3.1 — Демонстрація роботи сервера

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-11 4 курсу

Панченка С. В

```c
// ./lab_4/multiplex_server.c

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>

#include <unistd.h>
#include <signal.h>
#include <dirent.h>
#include <errno.h>

#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <sys/sendfile.h>

#include "client_utils.h"

#define MIN(x, y) (((x) < (y)) ? (x) : (y))

typedef enum {
    ClientStateTag_INVALID,
    ClientStateTag_RECEIVE_PROTOCOL_VERSION,
    ClientStateTag_SEND_MATCH_PROTOCOL_VERSION,
    ClientStateTag_RECEIVE_FILE_NAME,
    ClientStateTag_SEND_FILE_OPERATION_POSSIBILITY,
    ClientStateTag_SEND_FILE_SIZE,
    ClientStateTag_RECEIVE_CLIENT_READY,
    ClientStateTag_SEND_CHUNK,
    ClientStateTag_RECEIVE_FINISH,
} ClientStateTag;
```

```
typedef struct {
    ClientStateTag tag;
    union {
        struct ClientState_ReceiveProtocolVersion {
            int32_t client_fd;
        } receive_protocol_version;
        struct ClientState_SendMatchProtocolVersion {
            int32_t client_fd;
            uint8_t client_protocol_version;
        } send_match_protocol_version;
        struct ClientState_ReceiveFilename {
            int32_t client_fd;
        } receive_filename;
        struct ClientState_SendFileOperationPossibility {
            int32_t client_fd;
            int32_t fd;
        } send_file_operation_possibility;
        struct ClientState_SendChunkAndFileSize {
            int32_t client_fd;
            int32_t fd;
            off_t file_size;
        } send_file_size;
        struct ClientState_ReceiveClientReady {
            int32_t client_fd;
            int32_t fd;
            off_t file_size;
        } receive_client_ready;
        struct ClientState_SendChunk {
            int32_t client_fd;
            int32_t fd;
            off_t file_size;
            off_t file_offset;
        } send_chunk;
        struct ClientState_ReceiveFinish {
            int32_t client_fd;
        } receive_finish;
    } value;
```

```
} ClientState;

typedef uint16_t clients_count_t;

static ClientState construct_drop_connection(
    clients_count_t *const clients_count,
    const int32_t client_fd
) {
    printf("[client_fd: %d] [drop connection]\n", client_fd);

    --(*clients_count);
    checked_close(client_fd);
    ClientState state;
    state.tag = ClientStateTag_INVALID;
    return state;
}

static ClientState ClientState_transition(
    clients_count_t *const clients_count,
    const ClientState* const state,
    const fd_set* const readfds,
    const fd_set* const writefds,
    char* const filepath_buffer,
    const size_t filepath_buffer_offset
) {
    switch (state->tag) {
        case ClientStateTag_INVALID: {
            return *state;
        }
        case ClientStateTag_RECEIVE_PROTOCOL_VERSION: {
             const struct ClientState_ReceiveProtocolVersion* const
cur_state = &state->value.receive_protocol_version;
            if(!FD_ISSET(cur_state->client_fd, readfds)) {
                return *state;
            }
                                        printf("[client_fd:   %d]
[ClientStateTag_RECEIVE_PROTOCOL_VERSION]\n",           cur_state-
```

```
>client_fd);

            uint8_t client_protocol_version;
            if(
                            not  checked_read(cur_state->client_fd,
&client_protocol_version, sizeof(client_protocol_version), NULL)
            ) {
                    return construct_drop_connection(clients_count,
cur_state->client_fd);
            }

            ClientState new_state;
                                                    new_state.tag    =
ClientStateTag_SEND_MATCH_PROTOCOL_VERSION;
              new_state.value.send_match_protocol_version.client_fd
= cur_state->client_fd;
             new_state.value.send_match_protocol_version.client_pro
tocol_version = client_protocol_version;
            return new_state;
        }
        case ClientStateTag_SEND_MATCH_PROTOCOL_VERSION: {
                const struct ClientState_SendMatchProtocolVersion*
const cur_state = &state->value.send_match_protocol_version;
            if(!FD_ISSET(cur_state->client_fd, writefds)) {
                return *state;
            }
                                            printf("[client_fd:    %d]
[ClientStateTag_SEND_MATCH_PROTOCOL_VERSION]\n",         cur_state-
>client_fd);

                        const  bool  is_protocol_match  =  cur_state-
>client_protocol_version == PROTOCOL_VERSION;
                         if(not  checked_write(cur_state->client_fd,
&is_protocol_match, sizeof(is_protocol_match), NULL)) {
                    return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
```

```
            if(not is_protocol_match) {
                  return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
            ClientState new_state;
            new_state.tag = ClientStateTag_RECEIVE_FILE_NAME;
                     new_state.value.receive_filename.client_fd  =
cur_state->client_fd;
            return new_state;
        }
        case ClientStateTag_RECEIVE_FILE_NAME: {
                  const  struct  ClientState_ReceiveFilename*  const
cur_state = &state->value.receive_filename;
            if(!FD_ISSET(cur_state->client_fd, readfds)) {
                return *state;
            }
                                          printf("[client_fd:   %d]
[ClientStateTag_RECEIVE_FILE_NAME]\n", cur_state->client_fd);

                        if(not  checked_read(cur_state->client_fd,
filepath_buffer + filepath_buffer_offset, NAME_MAX, NULL)) {
                  return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
              filepath_buffer[filepath_buffer_offset + NAME_MAX] =
'\0';
                printf("[client_fd: %d] [filepath_buffer: %s]\n",
cur_state->client_fd, filepath_buffer);

            ClientState new_state;
                                                  new_state.tag    =
ClientStateTag_SEND_FILE_OPERATION_POSSIBILITY;
             new_state.value.send_file_operation_possibility.client
_fd = cur_state->client_fd;
              new_state.value.send_file_operation_possibility.fd =
open(filepath_buffer, O_RDONLY);
            return new_state;
```

```
        }
        case ClientStateTag_SEND_FILE_OPERATION_POSSIBILITY: {
             const struct ClientState_SendFileOperationPossibility
*const cur_state = &state->value.send_file_operation_possibility;
            if(!FD_ISSET(cur_state->client_fd, writefds)) {
                return *state;
            }
                                        printf("[client_fd:   %d]
[ClientStateTag_SEND_FILE_OPERATION_POSSIBILITY]\n",    cur_state-
>client_fd);

            bool is_possible = cur_state->client_fd != -1;
            if(not is_possible) {
                 checked_write(cur_state->client_fd, &is_possible,
sizeof(is_possible), NULL);
                   return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
            struct stat st;
            is_possible = fstat(cur_state->fd, &st) != -1;
            if(not is_possible) {
                checked_close(cur_state->fd);
                 checked_write(cur_state->client_fd, &is_possible,
sizeof(is_possible), NULL);
                   return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
                        if(not  checked_write(cur_state->client_fd,
&is_possible, sizeof(is_possible), NULL)) {
                checked_close(cur_state->fd);
                   return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
            ClientState new_state;
            new_state.tag = ClientStateTag_SEND_FILE_SIZE;
             new_state.value.send_file_size.client_fd = cur_state-
>client_fd;
```

```
            new_state.value.send_file_size.fd = cur_state->fd;
            new_state.value.send_file_size.file_size = st.st_size;
            return new_state;
        }
        case ClientStateTag_SEND_FILE_SIZE: {
              const struct ClientState_SendChunkAndFileSize *const
cur_state = &state->value.send_file_size;
            if(!FD_ISSET(cur_state->client_fd, writefds)) {
                return *state;
            }
                                          printf("[client_fd:   %d]
[ClientStateTag_SEND_FILE_SIZE]\n", cur_state->client_fd);

              printf("[client_fd: %d] [cur_state->file_size: %ld]\
n", cur_state->client_fd, cur_state->file_size);
                              const  uint32_t  network_file_size  =
htonl((uint32_t)cur_state->file_size);
                                 checked_write(cur_state->client_fd,
&network_file_size, sizeof(network_file_size), NULL);

            ClientState new_state;
            new_state.tag = ClientStateTag_RECEIVE_CLIENT_READY;
                  new_state.value.receive_client_ready.client_fd =
cur_state->client_fd;
              new_state.value.receive_client_ready.fd = cur_state-
>fd;
                  new_state.value.receive_client_ready.file_size =
cur_state->file_size;
            return new_state;
        }
        case ClientStateTag_RECEIVE_CLIENT_READY: {
               const struct ClientState_ReceiveClientReady *const
cur_state = &state->value.receive_client_ready;
            if(!FD_ISSET(cur_state->client_fd, readfds)) {
                return *state;
            }
                                          printf("[client_fd:   %d]
```

```
[ClientStateTag_RECEIVE_CLIENT_READY]\n", cur_state->client_fd);

            bool is_client_ready;
                        if(not  checked_read(cur_state->client_fd,
&is_client_ready, sizeof(is_client_ready), NULL)) {
                checked_close(cur_state->fd);
                    return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
                 printf("[client_fd: %d] [is_client_ready: %d]\n",
cur_state->client_fd, is_client_ready);
            if(not is_client_ready) {
                checked_close(cur_state->fd);
                    return construct_drop_connection(clients_count,
cur_state->client_fd);
            }
            ClientState new_state;
            new_state.tag = ClientStateTag_SEND_CHUNK;
                 new_state.value.send_chunk.client_fd = cur_state-
>client_fd;
            new_state.value.send_chunk.fd = cur_state->fd;
                 new_state.value.send_chunk.file_size = cur_state-
>file_size;
            new_state.value.send_chunk.file_offset = 0;
            return new_state;
        }
        case ClientStateTag_SEND_CHUNK: {
            ClientState new_generic_state = *state;
               struct ClientState_SendChunk *const new_cur_state =
&new_generic_state.value.send_chunk;
            if(!FD_ISSET(new_cur_state->client_fd, writefds)) {
                return new_generic_state;
            }
              printf("[client_fd: %d] [ClientStateTag_SEND_CHUNK]\
n", new_cur_state->client_fd);
             const off_t end_offset = MIN(new_cur_state->file_size,
new_cur_state->file_offset + CHUNK_SIZE);
```

```
            while(true) {
                    if(new_cur_state->file_size <= new_cur_state->file_offset) {
                    checked_close(new_cur_state->fd);
                                                new_generic_state.tag = ClientStateTag_RECEIVE_FINISH;
                    new_generic_state.value.receive_finish.client_fd = state->value.send_chunk.client_fd;
                    return new_generic_state;
                }
                            const off_t local_diff = end_offset - new_cur_state->file_offset;
                if(local_diff <= 0) {
                    break;
                }
                    // printf("[client_fd: %d] [pre new_cur_state->file_offset: %ld]\n", new_cur_state->client_fd, new_cur_state->file_offset);

                 const ssize_t nsendfile = sendfile(new_cur_state->client_fd, new_cur_state->fd, &new_cur_state->file_offset, (size_t)(local_diff));
                if(nsendfile == -1) {
                            printf("[Failed to sendfile] [errno: %d] [strerror: %s]\n", errno, strerror(errno));
                    break;
                }
                   // printf("[client_fd: %d] [post new_cur_state->file_offset: %ld]\n", new_cur_state->client_fd, new_cur_state->file_offset);
            }
                    // printf("[client_fd: %d] [post cycle]\n", new_cur_state->client_fd);
            return new_generic_state;
        }
        case ClientStateTag_RECEIVE_FINISH: {
```

```
                    const struct ClientState_ReceiveFinish *const
cur_state = &state->value.receive_finish;
            if(!FD_ISSET(cur_state->client_fd, readfds)) {
                return *state;
            }
                                        printf("[client_fd:   %d]
[ClientStateTag_RECEIVE_FINISH]\n", cur_state->client_fd);

            uint8_t signal_end_byte;
              checked_read(cur_state->client_fd, &signal_end_byte,
sizeof(signal_end_byte), NULL);

             printf("[client_fd: %d] [received signal_end_byte]\n",
cur_state->client_fd);

                    return construct_drop_connection(clients_count,
cur_state->client_fd);
        }
        default: {
            __builtin_unreachable();
        }
    }
}

static volatile sig_atomic_t keep_running = 1;
static       void       handle_sigint(const       int       value
__attribute_maybe_unused__) { keep_running = 0; }

static in_addr_t parse_address(const char *const value) {
    const in_addr_t address = inet_addr(value);
    ASSERT_POSIX(address);
    return address;
}
static in_port_t parse_port(const char *const value) {
    errno = 0;
    const uint64_t port = strtoul(value, NULL, 10);
    assert(errno == 0);
```

```
    assert(port == (uint64_t)((in_port_t)port));
    return htons((in_port_t)port);
}
static uint16_t parse_max_clients_count(const char *const value) {
    const uint64_t max_clients_count = strtoul(value, NULL, 10);
    assert(errno == 0);
      static const uint16_t FD_SETSIZE_ACCOUNTING_FOR_SERVER_FD =
FD_SETSIZE - 1;
                                assert(max_clients_count        <=
FD_SETSIZE_ACCOUNTING_FOR_SERVER_FD);
    return (uint16_t)max_clients_count;
}
static  int32_t  ClientState_client_fd(const  ClientState  *const
state) {
    switch(state->tag) {
        case ClientStateTag_INVALID: {
            __builtin_unreachable();
        }
        case ClientStateTag_RECEIVE_PROTOCOL_VERSION: {
                                                    return    state-
>value.receive_protocol_version.client_fd;
        }
        case ClientStateTag_SEND_MATCH_PROTOCOL_VERSION: {
                                                    return    state-
>value.send_match_protocol_version.client_fd;
        }
        case ClientStateTag_RECEIVE_FILE_NAME: {
            return state->value.receive_filename.client_fd;
        }
        case ClientStateTag_SEND_FILE_OPERATION_POSSIBILITY: {
                                                    return    state-
>value.send_file_operation_possibility.client_fd;
        }
        case ClientStateTag_SEND_FILE_SIZE: {
            return state->value.send_file_size.client_fd;
        }
        case ClientStateTag_RECEIVE_CLIENT_READY: {
```

```
            return state->value.receive_client_ready.client_fd;
        }
        case ClientStateTag_SEND_CHUNK: {
            return state->value.send_chunk.client_fd;
        }
        case ClientStateTag_RECEIVE_FINISH: {
            return state->value.receive_finish.client_fd;
        }
        default: {
            __builtin_unreachable();
        }
    }
}

static int init_file_descriptors(
    fd_set *readfds,
    fd_set *writefds,
    const int serverfd,
    const ClientState *const client_state_array,
    const clients_count_t max_clients_count
) {
    FD_ZERO(readfds);
    FD_ZERO(writefds);
    FD_SET(serverfd, readfds);
    int max_sd = serverfd;
    for(size_t i = 0; i < max_clients_count; ++i) {
        const ClientState* state = &client_state_array[i];
        if(state->tag != ClientStateTag_INVALID) {
            const int client_fd = ClientState_client_fd(state);
            FD_SET(client_fd, readfds);
            FD_SET(client_fd, writefds);
            if(client_fd > max_sd) {
                max_sd = client_fd;
            }
        }
    }
    return max_sd;
```

```
}

int main(
    const int argc,
    const char* const argv[]
) {
    {
        struct sigaction sa;
        sa.sa_handler = handle_sigint;
        ASSERT_POSIX(sigemptyset(&sa.sa_mask));
        sa.sa_flags = 0;
        ASSERT_POSIX(sigaction(SIGINT, &sa, NULL));
    }
    assert(argc == 5);

          const int listenfd = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);
    ASSERT_POSIX(listenfd);
    {
        struct sockaddr_in srv_sin4 = {
            .sin_family  = AF_INET,
            .sin_addr.s_addr = parse_address(argv[1]),
            .sin_port = parse_port(argv[2]),
        };
         ASSERT_POSIX(bind(listenfd, (struct sockaddr *)&srv_sin4,
sizeof(srv_sin4)));
        static const uint8_t MAX_BACKLOG = 10;
        ASSERT_POSIX(listen(listenfd, MAX_BACKLOG));
    }
    char filepath_buffer[PATH_MAX];
    uint16_t filepath_buffer_offset;
    {
        static const uint8_t slash_character_addition = 1;
              const size_t len_with_slash = strlen(argv[3]) +
slash_character_addition;
        assert((len_with_slash + NAME_MAX) <= PATH_MAX);
        filepath_buffer_offset = (uint16_t)len_with_slash;
```

```
        strcpy(filepath_buffer, argv[3]);
        strcat(filepath_buffer, "/");
    }

                    const     uint16_t     max_clients_count     =
parse_max_clients_count(argv[4]);

    fd_set readfds, writefds;
                ClientState    *const    client_state_array    =
calloc(max_clients_count, sizeof(ClientState));
    clients_count_t client_state_array_count = 0;
    for(size_t i = 0; i < max_clients_count; ++i) {
        client_state_array[i].tag = ClientStateTag_INVALID;
    }

    while(keep_running) {
        // printf("[main cycle start]\n");
        const int max_fd = init_file_descriptors(
                &readfds, &writefds, listenfd, client_state_array,
max_clients_count
        );
        // printf("[pre select] [max_fd: %d]\n", max_fd);
         if(select(max_fd + 1, &readfds, &writefds, NULL, NULL) ==
-1) {
                // printf("[select] [errno: %d] [strerror: %s]\n",
errno, strerror(errno));
            continue;
        }
        // printf("[post select] [max_fd: %d]\n", max_fd);
                            if(FD_ISSET(listenfd,   &readfds)   &&
client_state_array_count < max_clients_count) {
            struct sockaddr_in address;
            socklen_t addr_len = sizeof(address);
            // printf("[pre accept]\n");
                  const  int  client_fd  =  accept(listenfd,  (struct
sockaddr *)&address, &addr_len);
            // printf("[post accept]\n");
```

```
            if(client_fd == -1) {
                // printf("[accept] [errno: %d] [strerror: %s]\n",
errno, strerror(errno));
                continue;
            }
                printf("[New connection] [client_fd: %d] [IP: %s]
[port: %d]\n",
                            client_fd,  inet_ntoa(address.sin_addr),
ntohs(address.sin_port));

            for(size_t i = 0; i < max_clients_count; ++i) {
                ClientState* state = &client_state_array[i];
                if(state->tag == ClientStateTag_INVALID) {
                                                    state->tag   =
ClientStateTag_RECEIVE_PROTOCOL_VERSION;
                                                                state-
>value.receive_protocol_version.client_fd = client_fd;
                    ++client_state_array_count;
                    break;
                }
            }
        }

        for(size_t i = 0; i < max_clients_count; ++i) {
            ClientState* state = &client_state_array[i];
                                                        *state    =
ClientState_transition(&client_state_array_count, state, &readfds,
&writefds, filepath_buffer, filepath_buffer_offset);
        }
        // printf("[main cycle end]\n");
    }

    free(client_state_array);
    assert(checked_close(listenfd));
    return EXIT_SUCCESS;
}
```

```c
// ./lab_4/client.c

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <netinet/in.h>
#include "client_utils.h"

typedef struct {
    const char *address;
    uint16_t port;
    const char *filename;
    size_t max_file_size;
} ClientConfig;

static void print_config(const ClientConfig *config) {
    printf("Client Configuration:\n");
    printf("\tAddress: %s\n", config->address);
    printf("\tPort: %d\n", config->port);
    printf("\tFilename: %s\n", config->filename);
    printf("\tMaximum file size: %ld\n", config->max_file_size);
}

static ClientConfig handle_cmd_args(const int argc, char **argv) {
    if (argc != 5) {
         fprintf(stderr, "Usage: %s <server_address> <server_port>
<filename> <max_file_size>\n", argv[0]);
        exit(1);
    }
```

```
    const ClientConfig config = {
        .address = argv[1],
        .port = (uint16_t)atoi(argv[2]),
        .filename = argv[3],
        .max_file_size = (uint64_t)atoi(argv[4])
    };
    print_config(&config);
    return config;
}

static void receive_file(
    const int sock,
    const size_t file_size,
    const int pipe_in,
    const int pipe_out,
    const int file_fd
) {
    printf("[Started receiving file file]\n");
    size_t nread = 0;
    off_t write_offset = 0;
    while((size_t)write_offset < file_size) {
        {
                const ssize_t local_read = splice(sock, NULL,
pipe_out, NULL, file_size - nread, 0);
            if(local_read < 0) {
                    printf("[Failed to read splice] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
                break;
            }
            nread += (size_t)local_read;
        }
        {
                const ssize_t local_write = splice(pipe_in, NULL,
file_fd, &write_offset, file_size - (size_t)write_offset, 0);
            if(local_write < 0) {
                    printf("[Failed to write splice] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
```

```
                break;
            }
        }
    }
    printf("[Finished receiving file file]\n");
}

static void main_logic(const ClientConfig *const config, const int
sock) {
    {
        struct sockaddr_in server_addr;
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(config->port);
                    if   (inet_pton(AF_INET,   config->address,
&server_addr.sin_addr) <= 0) {
            printf("[Failed inet_pton] [errno: %d] [strerror: %s]\
n", errno, strerror(errno));
            return;
        }
             if  (connect(sock,  (struct  sockaddr  *)&server_addr,
sizeof(server_addr)) < 0) {
               printf("[Connection failed] [errno: %d] [strerror:
%s]\n", errno, strerror(errno));
            return;
        }
    }
        const uint8_t protocol_version = PROTOCOL_VERSION;
                  if(not  checked_write(sock,  &protocol_version,
sizeof(protocol_version), NULL)) {
             printf("[Failed to send protocol version] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
            return;
        }
                  printf("[Written  protocol  version:  %d]\n",
PROTOCOL_VERSION);
        {
            bool is_protocol_version_ok;
```

```
            if(not checked_read(sock, &is_protocol_version_ok,
sizeof(is_protocol_version_ok), NULL)) {
                printf("[Failed to receive protocol version ok]
[errno: %d] [strerror: %s]\n", errno, strerror(errno));
                return;
            }
            if(not is_protocol_version_ok) {
                printf("[Protocol version mismatch]\n");
                return;
            }
            printf("[Protocol version match]\n");
        }
        char filename_buffer[NAME_MAX];
                    strncpy(filename_buffer,  config->filename,
ARRAY_SIZE(filename_buffer));
                  if(not  checked_write(sock,  filename_buffer,
ARRAY_SIZE(filename_buffer), NULL)) {
             printf("[Failed to send filename buffer] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
            return;
        }
          printf("[Send filename_buffer] [filename_buffer: %s]\n",
filename_buffer);
        {
            bool is_file_operation_possible;
             if(not checked_read(sock, &is_file_operation_possible,
sizeof(is_file_operation_possible), NULL)) {
                                    printf("[Failed  to  receive
is_file_operation_possible] [errno: %d] [strerror: %s]\n", errno,
strerror(errno));
                return;
            }
                    printf("[is_file_operation_possible:  %d]\n",
is_file_operation_possible);
            if(not is_file_operation_possible) {
                return;
            }
```

```
        }
        uint32_t file_size;
          if(not checked_read(sock, &file_size, sizeof(file_size),
NULL)) {
               printf("[Failed to receive file size] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
            return;
        }
        file_size = ntohl(file_size);

    printf("[File size: %u]\n", file_size);
    {
             const bool is_client_ready = file_size <= config-
>max_file_size;
                  if(not checked_write(sock, &is_client_ready,
sizeof(is_client_ready), NULL)) {
              printf("[Failed to send is_client_ready: %d] [errno:
%d] [strerror: %s]\n", is_client_ready, errno, strerror(errno));
        }
        if(not is_client_ready) {
                printf("[Server file size is too large: %u]\n",
file_size);
            return;
        }
    }
    // while(true) {}
    int pipefd[2];
    if(pipe(pipefd) < 0) {
         printf("[Can not create pipe] [errno: %d] [strerror: %s]\
n", errno, strerror(errno));
        const bool is_client_ready = false;
                  if(not checked_write(sock, &is_client_ready,
sizeof(is_client_ready), NULL)) {
              printf("[Failed to send is_client_ready: %d] [errno:
%d] [strerror: %s]\n", is_client_ready, errno, strerror(errno));
        }
    } else {
```

```
            const int file_fd = open(config->filename, O_WRONLY |
O_CREAT | O_TRUNC, 0644);
        if(file_fd < 0) {
            const bool is_client_ready = false;
             printf("[Failed to open file for writing] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
                    if(not checked_write(sock, &is_client_ready,
sizeof(is_client_ready), NULL)) {
                    printf("[Failed to send is_client_ready: %d]
[errno: %d] [strerror: %s]\n", is_client_ready, errno,
strerror(errno));
            }
        } else {
            const bool is_client_ready = true;
                    if(not checked_write(sock, &is_client_ready,
sizeof(is_client_ready), NULL)) {
                    printf("[Failed to send is_client_ready: %d]
[errno: %d] [strerror: %s]\n", is_client_ready, errno,
strerror(errno));
            } else {
                        receive_file(sock, file_size, pipefd[0],
pipefd[1], file_fd);
                uint8_t signal_end_byte;
                            checked_write(sock, &signal_end_byte,
sizeof(signal_end_byte), NULL);
            }
            if(not checked_close(file_fd)) {
                     printf("[Failed to close file] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
            }
        }
        if(not checked_close(pipefd[0])) {
                printf("[Failed to close pipe 0] [errno: %d]
[strerror: %s]\n", errno, strerror(errno));
        }
        if(not checked_close(pipefd[1])) {
                printf("[Failed to close pipe 1] [errno: %d]
```

```
[strerror: %s]\n", errno, strerror(errno));
        }
    }
}

int main(const int argc, char *argv[]) {
    const ClientConfig config = handle_cmd_args(argc, argv);

    const int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Socket creation failed");
    } else {
        main_logic(&config, sock);
    }
    return EXIT_SUCCESS;
}


// ./lab_4/client_utils.h

#pragma once
#define _GNU_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <inttypes.h>
#include <unistd.h>
#include <sys/socket.h>
#include <errno.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <iso646.h>
#include <assert.h>
#include <stdbool.h>
#include <linux/limits.h>
```

```
#define ALWAYS_INLINE static inline __attribute((always_inline))

#define CONCAT_IMPL(x, y) x##y
#define CONCAT(x, y) CONCAT_IMPL(x, y)

#define UNIQUE_NAME_LINE(prefix) CONCAT(prefix, __LINE__)
#define UNIQUE_NAME_COUNTER(prefix) CONCAT(prefix, __COUNTER__)
#define                                          UNIQUE_NAME(prefix)
UNIQUE_NAME_COUNTER(UNIQUE_NAME_LINE(prefix))

#define ARRAY_SIZE(data) (sizeof((data)) / sizeof(data[0]))

#define    ASSERT_POSIX(expression)    assert((expression)    !=
(__typeof__((expression)))-1)

static bool checked_read(const int fd, void *const vptr, const
size_t n, size_t *nread) {
    if(nread == NULL) {
        nread = alloca(sizeof(*nread));
    }
    *nread = 0;
    while(*nread < n) {
         ssize_t local_nread = read(fd, (char*)vptr + *nread, n -
*nread);
        // printf("local_nread: %lu\n", local_nread);
        if(local_nread < 0) {
            if(errno == EINTR) {
                continue;
            }
            return false;
        } else if(local_nread == 0) {
            // EOF
            break;
        }
        *nread += (size_t)local_nread;
    }
    return true;
```

```
}

static bool checked_write(const int fd, const void *vptr, const
size_t n, size_t *nwrite) {
    if(nwrite == NULL) {
        nwrite = alloca(sizeof(*nwrite));
    }
    *nwrite = 0;
    while(*nwrite < n) {
            ssize_t local_nwrite = write(fd, (const char*)vptr +
*nwrite, n - *nwrite);
        if(local_nwrite <= 0) {
            if (local_nwrite < 0 and errno == EINTR) {
                continue;
            }
            return false;
        }
        *nwrite += (size_t)local_nwrite;
    }
    return true;
}

static bool checked_close(const int fd) {
    while(close(fd) == -1) {
        if(errno == EINTR) {
            continue;
        }
        return false;
    }
    return true;
}

static const uint8_t PROTOCOL_VERSION = 17;
static const uint16_t CHUNK_SIZE = 100;
```