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 №3

Мережеве програмування в середовищі Unix

Тема: Багатопроцесний ітеративний TCP клієнт-сервер

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

Перевірив:
Сімоненко А.В.

Київ 2025

ЗМІСТ

1 Мета лабораторної роботи.....6

2 Завдання.....7

3 Виконання.....9

3.1 Ітеративний сервер.....9

3.2 Паралельний сервер.....12

3.3 Сервер з пулом потоків.....14

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ.....16

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Реалізувати простий мережевий сервіс передачі файлів за допомогою протоколу TCP.

2 ЗАВДАННЯ

Розробити клієнт та сервер, які виконують наступне:

1. Клієнт підтримує такі аргументи командного рядка: адреса сервера, порт сервера, ім’я файлу, максимальний розмір файлу.
2. Сервер підтримує такі аргументи командного рядка: адреса сервера, порт сервера, шляхове ім’я директорії.
3. Клієнт та сервер використовують транспортний протокол TCP для мережевого з’єднання.
4. Клієнт відправляє запит серверу з ім’ям файлу, яке вказано в аргументі його командного рядка. Сервер отримує запит від клієнта, шукає звичайний файл з вказаним ім’ям у директорії, шляхове ім’я якої вказано в аргументі його командного рядка, та відправляє клієнту вміст файлу. Клієнт записує отриманий вміст у звичайний файл.

Протокол рівня застосунку має наступні характеристики:

1. Протокол має версію. Якщо версії протоколів, які використовують клієнт та сервер не збігаються, тоді з’єднання між клієнтом та сервером треба завершити.
2. Ім’я файлу в запиті клієнта повинно складатися з символів ASCII, які дозволені для імені файлу в наявній ФС (літери, цифри, знаки пунктуації і т. ін.). Максимальна довжина іменіфайлу обмежена. Клієнт може надіслати які-завгодно дані замість імені файлу, сервер має перевірити коректність цих даних. Клієнт має надіслати ім’я файлу, а не шляхове ім’я.
3. Сервер відправляє клієнту інформацію чи було знайдено файл з вказаним ім’ям та його розмір, у випадку помилки серер відправляє клієнту номер помилки та завершує з’єднання. Розмір файлу не перевищує значення (264 - 1) байт (тобто є 64 біт для значення розміру файлу в заголовку). У випадку помилки клієнт виводить інформацію про помилку та завершує з’єднання. Якщо розмір файлу перевищує вказаний максимальний розмір файлу в аргументі командного рядка клієнта, тоді клієнт відправляє серверу

повідомлення про відмову отримувати вміст файлу, інакше клієнт відправляє серверу повідомлення про готовність отримувати вміст файлу.

4. Якщо сервер отримав повідомлення від клієнта про готовність отримувати вміст файлу, тоді він відправляє вміст файлу частинами (тобто може потребуватися кілька викликів відповідного системного виклику для відправлення вмісту файлу). Розмір частини визначається в сервері константним значенням. Відправивши весь вміст файлу, сервер завершує з’єднання. Отримавши весь вміст файлу клієнт завершує з’єднання.

Треба реалізувати наступні реалізації серверів:

1. Ітеративний сервер, який опрацьовує запити одного клієнта повністю, перед тим, як почати опрацьовувати запити наступного клієнта.
2. Паралельний сервер, який створює нові процеси для опрацювання запитів нових клієнтів. Сервер має обмеження на максимальну кількість дочірніх процесів, які опрацьовують запити клієнтів. Ця максимальна кількість вказується в аргументі командного рядка сервера. Сервер не приймає нові TCP з’єднання від клієнтів після досягнення цієї кількості.
3. Паралельний сервер, який заздалегідь створює нові процеси для опрацювання запитів клієнтів, кожний дочірній процес є ітеративним сервером, як у першому пункті. Кількість дочірніх процесів, які має створити сервер, вказується в аргументі командного рядка сервера. Сервер не має завершувати своє виконання у випадку виникнення несистемної помилки. Для перевірки коректності роботи програм рекомендується виводити повідомлення про дії в програмах (адреси, номери портів, вміст заголовків).

3 ВИКОНАННЯ

3.1 Ітеративний сервер

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 / 'iterative_server.o'
ADDRESS  = '0.0.0.0'
PORT = '55002'
MAX_FILE_SIZE = '1000000000'

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

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

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 ітеративного сервера, до якого звертаються клієнти.

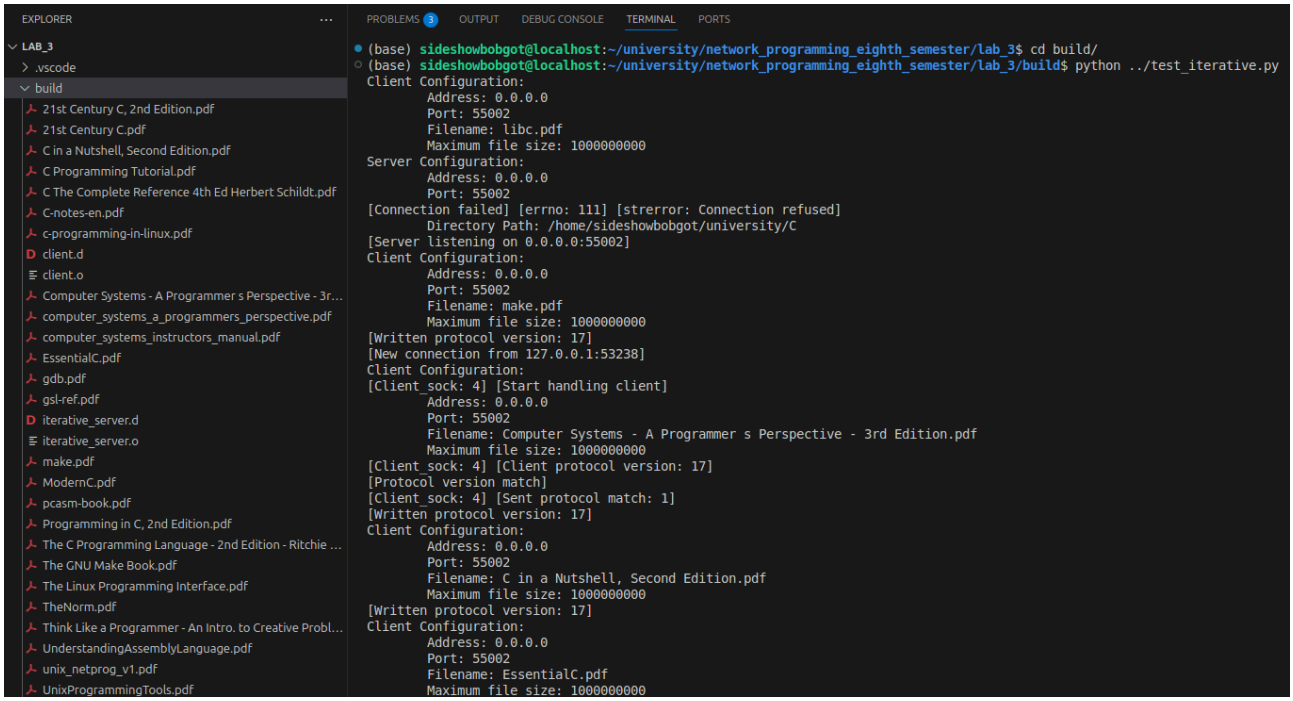

Рисунок 3.1 — Демонстрація роботи сервера

Сервер по черзі опрацьовує запит від кожного клієнта, як бачиму на логах нижче:

```
[New connection from 127.0.0.1:53240]
[Client_sock: 4] [Start handling client]
[Client_sock: 4] [Client protocol version: 17]
[Client_sock: 4] [Sent protocol match: 1]
[Protocol version match]
[Finished receiving file file]
[File size: 37713106]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[New connection from 127.0.0.1:53242]
[Client_sock: 4] [Start handling client]
[Client_sock: 4] [Client protocol version: 17]
[Client_sock: 4] [Sent protocol match: 1]
[Protocol version match]
[Finished receiving file file]
[File size: 10557760]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
```

```
[Client_sock: 4] [Finished sending file]
[New connection from 127.0.0.1:53256]
[Client_sock: 4] [Start handling client]
[Client_sock: 4] [Client protocol version: 17]
[Client_sock: 4] [Sent protocol match: 1]
[Protocol version match]
[Finished receiving file file]
[File size: 87101]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[New connection from 127.0.0.1:53264]
[Client_sock: 4] [Start handling client]
[Client_sock: 4] [Client protocol version: 17]
[Client_sock: 4] [Sent protocol match: 1]
[Protocol version match]
[Finished receiving file file]
[File size: 7079105]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
```

3.2 Паралельний сервер

Аналогічно до пункту 3.1 на рисунку 3.2 виконаємо тестування паралельного сервера. Як ми бачимо логи перемішані, що вказує на те, що багато процесів вписують в stdout.

```
[Started receiving file file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Finished receiving file file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[File size: 21298031]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
```

```
[Client_sock: 4] [Finished sending file]
[File size: 9995457]
[File size: 8797200]
[Finished receiving file file]
[Started receiving file file]
[Started receiving file file]
[File size: 130791]
[Client_sock: 4] [Ready to send file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[Client_sock: 4] [Ready to send file]
[Started receiving file file]
[File size: 5555005]
[Started receiving file file]
[Client_sock: 4] [Finished sending file]
[Client_sock: 4] [Finished sending file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[Finished receiving file file]
[Client_sock: 4] [Finished sending file]
[Finished receiving file file]
[Finished receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[Finished receiving file file]
[Finished receiving file file]
[Client_sock: 4] [Finished sending file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
```

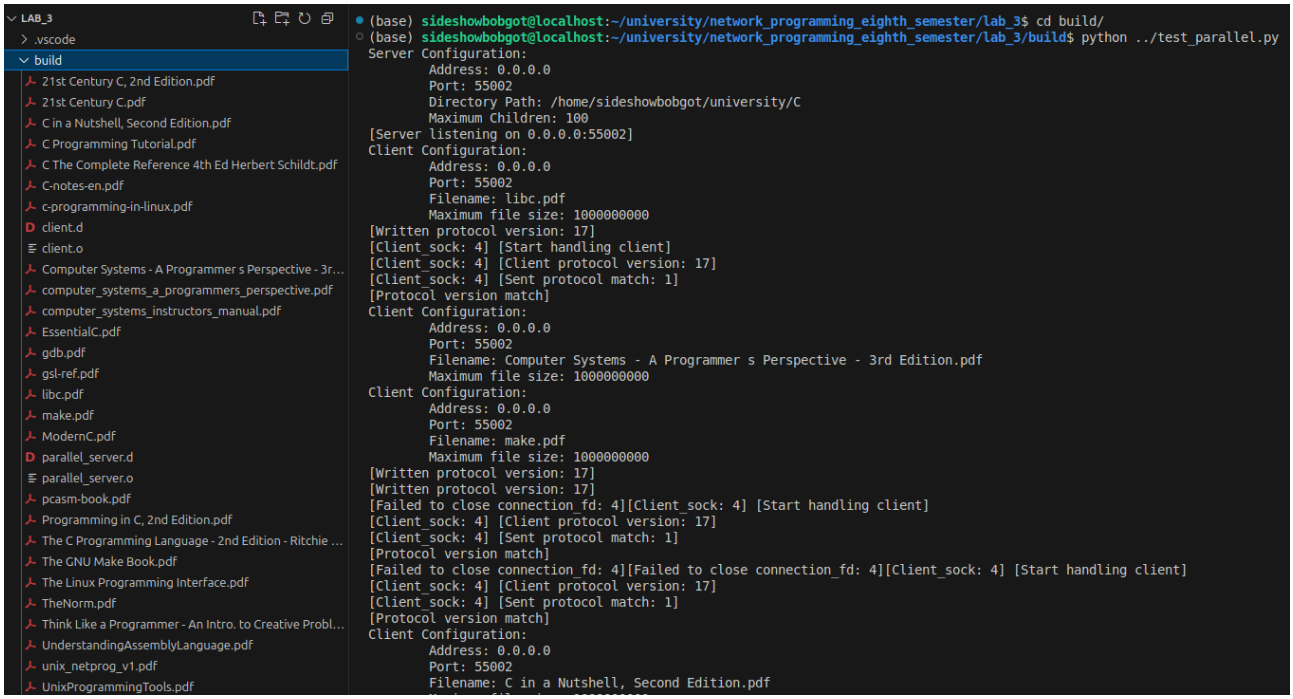

Рисунок 3.2 — Демонстрація роботи сервера

3.3 Сервер з пулом потоків

Аналогічно до пункту 3.1 на рисунку 3.3 виконаємо тестування паралельного сервера. Як ми бачимо логи перемішані, що вказує на те, що багато процесів вписують в stdout. Як ми бачимо логи перемішані, що вказує на те, що багато процесів вписують в stdout.

```
[Finished receiving file file]
[File size: 613098]
[File size: 37713106]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[File size: 13186934]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[File size: 7079105]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Finished receiving file file]
```

```
[File size: 3130505]
[Finished receiving file file]
[Started receiving file file]
[File size: 5797785]
[File size: 6374486]
[Client_sock: 4] [Ready to send file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Finished sending file]
[File size: 5964461]
[File size: 7758048]
[Started receiving file file]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
[Client_sock: 4] [Ready to send file]
[File size: 3497031]
[Started receiving file file]
[Client_sock: 4] [Ready to send file]
```

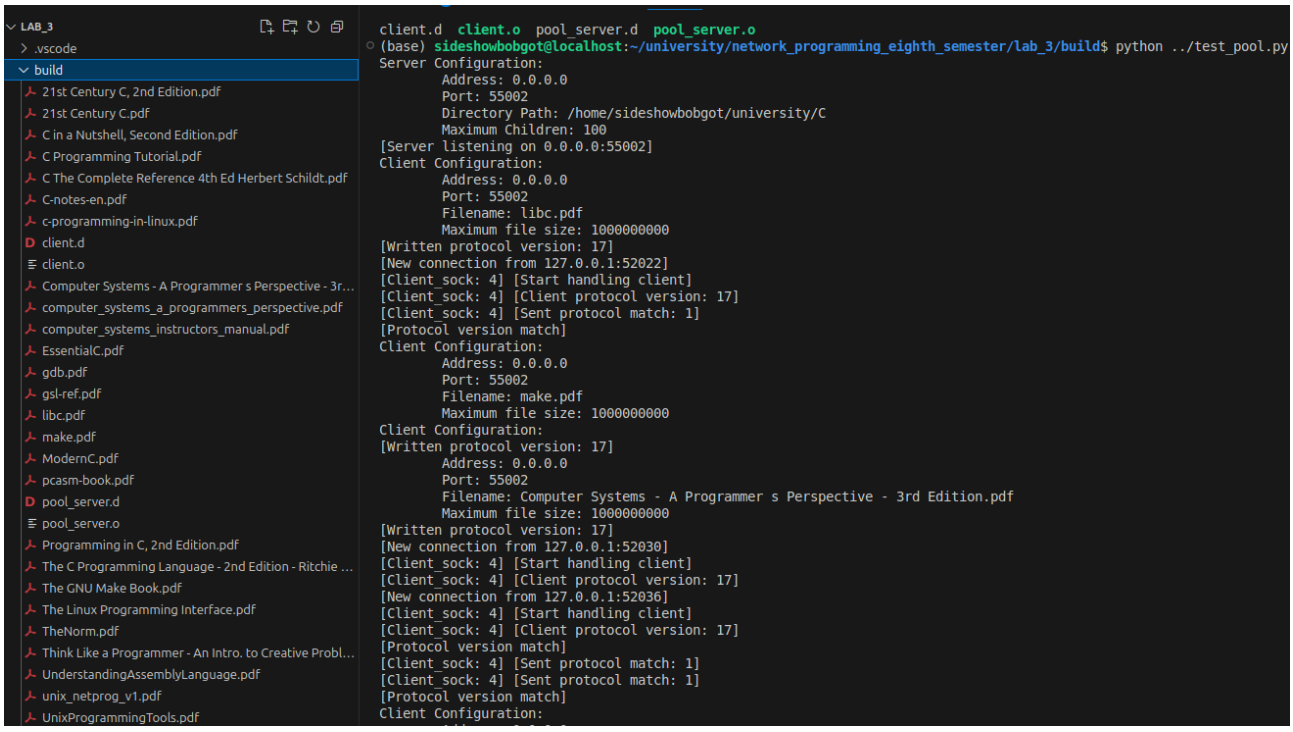

Рисунок 3.3 — Демонстрація роботи сервера

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

```
// ./lab_3/iterative_server_utils_two.h

#include "iterative_server_utils_one.h"

static void iterative_server_main_loop(
    const int listenfd,
    const char *const dir_path
) {
    while (keep_running) {
        struct sockaddr_in client_in;
        socklen_t addrlen = sizeof(client_in);
             const int connection_fd = accept(listenfd, (struct
sockaddr *)&client_in, &addrlen);
        if (connection_fd < 0) {
            continue;
        }
                    printf("[New connection from %s:%d]\n",
inet_ntoa(client_in.sin_addr), ntohs(client_in.sin_port));
        handle_client(connection_fd, dir_path);
        if(not checked_close(connection_fd)) {
              printf("[Failed to close client connection: %d]\n",
connection_fd);
        }
    }
}


// ./lab_3/iterative_server.c

#include "iterative_server_utils_two.h"

#include <signal.h>

static IterativeServerConfig handle_cmd_args(const int argc, char
**argv) {
    if (argc != 4) {
         fprintf(stderr, "Usage: %s <server_address> <server_port>
```

```
<directory_path>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const IterativeServerConfig config = {
        .address = argv[1],
        .port = (uint16_t)atoi(argv[2]),
        .dir_path = argv[3],
    };

    iterative_server_print_config(&config);

    return config;
}

static void inner_function(const int listenfd, const
IterativeServerConfig *const config) {
    struct sockaddr_in srv_sin4 = {
        .sin_family  = AF_INET,
        .sin_port = htons(config->port),
        .sin_addr.s_addr = inet_addr(config->address)
    };
       ASSERT_POSIX(bind(listenfd, (struct sockaddr *)&srv_sin4,
sizeof(srv_sin4)));
    ASSERT_POSIX(listen(listenfd, MAX_BACKLOG));

       printf("[Server listening on %s:%d]\n", config->address,
config->port);
    iterative_server_main_loop(listenfd, config->dir_path);
}

int main(const int argc, char *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));
    }

      const IterativeServerConfig config = handle_cmd_args(argc,
argv);
          const int listenfd = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);
    ASSERT_POSIX(listenfd);
    inner_function(listenfd, &config);
    assert(checked_close(listenfd));
    return EXIT_SUCCESS;
}


// ./lab_3/parallel_server.c

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <signal.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <errno.h>
#include <err.h>
#include <stdatomic.h>
#include "iterative_server_utils_one.h"

typedef struct {
    IterativeServerConfig config;
    int32_t max_children;
} ParallelServerConfig;

static          void          parallel_server_print_config(const
ParallelServerConfig *config) {
    iterative_server_print_config(&config->config);
    printf("\tMaximum Children: %d\n", config->max_children);
```

```
}

static ParallelServerConfig handle_cmd_args(const int argc, const
char **argv) {
    if (argc != 5) {
                printf("Usage: %s <server_address> <server_port>
<directory_path> <max_children>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const ParallelServerConfig config = {
        .config.address = argv[1],
        .config.port = (uint16_t)atoi(argv[2]),
        .config.dir_path = argv[3],
        .max_children = atoi(argv[4])
    };
    assert(config.max_children > 0);
    parallel_server_print_config(&config);
    return config;
}

static atomic_int active_children = 0;

static void wait_finish_child_processes(void) {
    while (true) {
        const pid_t pid = waitpid(-1, NULL, WNOHANG);
        switch (pid) {
            case -1: {
                if (errno != ECHILD) {
                    printf("[Failed waitpid]");
                }
                return;
            }
            case 0: {
                return;
            }
            default: {
```

```
                printf("[Process %jd exited]\n", (intmax_t)pid);
                atomic_fetch_add(&active_children, -1);
            }
        }
    }
}

static void handle_sigchld(const int value __attribute__((unused))) {
    wait_finish_child_processes();
}

static void socketfd_valid(const ParallelServerConfig *config, const int socketfd) {
    {
        const struct sockaddr_in srv_sin4 = {
            .sin_family  = AF_INET,
            .sin_port = htons(config->config.port),
            .sin_addr.s_addr = inet_addr(config->config.address)
        };
              ASSERT_POSIX(bind(socketfd, (const struct sockaddr *)&srv_sin4, sizeof(srv_sin4)));
    }
    ASSERT_POSIX(listen(socketfd, MAX_BACKLOG));
            printf("[Server listening on %s:%d]\n", config->config.address, config->config.port);

    sigset_t sigcld_unblock_mask;
    sigprocmask(SIG_BLOCK, NULL, &sigcld_unblock_mask);
    sigdelset(&sigcld_unblock_mask, SIGCLD);

    sigset_t sigcld_block_mask;
    sigprocmask(SIG_BLOCK, NULL, &sigcld_block_mask);
    sigaddset(&sigcld_block_mask, SIGCHLD);

    sigprocmask(SIG_BLOCK, &sigcld_block_mask, NULL);
```

```
    while(keep_running) {
        const int connection_fd = accept(socketfd, NULL, NULL);
        if(connection_fd < 0) {
            continue;
        }
        const pid_t pid = fork();
        if(pid < 0) {
            perror("[Failed to fork]");
        } else if (pid == 0) {
            handle_client(connection_fd, config->config.dir_path);
            return;
        } else {
            atomic_fetch_add(&active_children, 1);
            while(atomic_fetch_add(&active_children, 0) == config-
>max_children) {
                sigsuspend(&sigcld_unblock_mask);
            }
        }
        if(not close(connection_fd)) {
                printf("[Failed to close connection_fd: %d]",
connection_fd);
        }
    }
    wait_finish_child_processes();
}

int main(const int argc, const char *argv[]) {
    {
        struct sigaction sa;
        ASSERT_POSIX(sigemptyset(&sa.sa_mask));
        sa.sa_flags = 0;

        sa.sa_handler = handle_sigint;
        ASSERT_POSIX(sigaction(SIGINT, &sa, NULL));

        sa.sa_handler = handle_sigchld;
        ASSERT_POSIX(sigaction(SIGCLD, &sa, NULL));
```

```
    }
       const ParallelServerConfig config = handle_cmd_args(argc,
argv);
          const int listenfd = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);
    ASSERT_POSIX(listenfd);
    socketfd_valid(&config, listenfd);
    assert(checked_close(listenfd));
    return EXIT_SUCCESS;
}


// ./lab_3/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 size_t file_size = ({
        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");
        }
        filename_buff_t filename_buffer;
                      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;
        }
        {
            bool is_file_size_ok;
                      if(not  checked_read(sock,  &is_file_size_ok,
sizeof(is_file_size_ok), NULL)) {
```

```
                    printf("[Failed to receive is_file_size_ok]
[errno: %d] [strerror: %s]\n", errno, strerror(errno));
                return;
            }
            if(not is_file_size_ok) {
                printf("[File size is not ok]\n");
                return;
            }
        }
        size_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;
        }
        be64toh(file_size);
    });
    printf("[File size: %ld]\n", file_size);
    {
             const bool is_client_ready = file_size <= config-
>max_file_size;
        if(not is_client_ready) {
                printf("[Server file size is too large: %lu]\n",
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));
            }
            return;
        }
    }
    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);
            }
            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_3/iterative_server_utils_one.h

#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <dirent.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```c
#include <stdbool.h>
#include <fcntl.h>
#include <sys/sendfile.h>
#include <endian.h>
#include <alloca.h>
#include <signal.h>

#include "client_utils.h"

typedef struct {
    const char *address;
    uint16_t port;
    const char *dir_path;
} IterativeServerConfig;

static void iterative_server_print_config(const
IterativeServerConfig *config) {
    printf("Server Configuration:\n");
    printf("\tAddress: %s\n", config->address);
    printf("\tPort: %d\n", config->port);
    printf("\tDirectory Path: %s\n", config->dir_path);
}

static volatile sig_atomic_t keep_running = true;
static void handle_sigint(const int val __attribute__((unused))) {
    keep_running = false;
}

enum {
    MAX_BACKLOG = 10
};

static void with_file_open(
    const int fd,
    const int client_sock,
    const char *const buffer
) {
```

```
    struct stat st;
    if(fstat(fd, &st) == -1) {
          printf("[Client_sock: %d] [Error stat: %s] [errno: %d] [strerror: %s]\n", client_sock, buffer, errno, strerror(errno));
        const bool is_file_size_ok = false;
             if(not checked_write(client_sock, &is_file_size_ok, sizeof(is_file_size_ok), NULL)) {
             printf("[Client_sock: %d] [Failed to inform file size not ok: %s] [errno: %d] [strerror: %s]\n", client_sock, buffer, errno, strerror(errno));
        }
        return;
    }
    {
        const bool is_file_size_ok = true;
             if(not checked_write(client_sock, &is_file_size_ok, sizeof(is_file_size_ok), NULL)) {
              printf("[Client_sock: %d] [Failed to inform failure file size ok] [errno: %d] [strerror: %s]\n", client_sock, errno, strerror(errno));
            return;
        }
    }
    {
                          const size_t network_file_size = htobe64((size_t)st.st_size);
            if(not checked_write(client_sock, &network_file_size, sizeof(network_file_size), NULL)) {
             printf("[Client_sock: %d] [Failed to send file size] [errno: %d] [strerror: %s]\n", client_sock, errno, strerror(errno));
            return;
        }
    }
    bool is_client_ready;
            if(not checked_read(client_sock, &is_client_ready, sizeof(is_client_ready), NULL)) {
```

```
        printf("[Client_sock: %d] [Failed to receive clients file approval] [errno: %d] [strerror: %s]\n", client_sock, errno, strerror(errno));
        return;
    }
    if(not is_client_ready) {
            printf("[Client_sock: %d] [Client rejected file receiving]\n", client_sock);
        return;
    }

        printf("[Client_sock: %d] [Ready to send file]\n", client_sock);
    {
        off_t offset = 0;
        while(offset < st.st_size) {
            const ssize_t nsendfile = sendfile(client_sock, fd, &offset, (size_t)(st.st_size - offset));
            if(nsendfile < 0) {
                printf("[Client_sock: %d] [Failed to sendfile] [errno: %d] [strerror: %s]\n", client_sock, errno, strerror(errno));
                return;
            } else if(nsendfile == 0) {
                break;
            }
        }
    }
        printf("[Client_sock: %d] [Finished sending file]\n", client_sock);
}

static void handle_client(
    const int client_sock,
    const char* const dir_path
) {
    {
```

```
            printf("[Client_sock: %d] [Start handling client]\n",
client_sock);
        uint8_t client_protocol_version;
        if(not checked_read(client_sock, &client_protocol_version,
sizeof(client_protocol_version), NULL)) {
            printf("[Client_sock: %d] [Failed to receive request]
[errno: %d] [strerror: %s]\n", client_sock, errno,
strerror(errno));
            return;
        }
         printf("[Client_sock: %d] [Client protocol version: %d]\
n", client_sock, client_protocol_version);
        {
            const bool is_protocol_match = client_protocol_version
== PROTOCOL_VERSION;
            if(not checked_write(client_sock, &is_protocol_match,
sizeof(is_protocol_match), NULL)) {
                printf("[Client_sock: %d] [Failed to send protocol
match] [errno: %d] [strerror: %s]\n", client_sock, errno,
strerror(errno));
                return;
            }
             printf("[Client_sock: %d] [Sent protocol match: %d]\
n", client_sock, is_protocol_match);
            if(not is_protocol_match) {
                return;
            }
        }
    }
    char *buffer;
    {
        filename_buff_t filename_buffer;
               if(not checked_read(client_sock, filename_buffer,
ARRAY_SIZE(filename_buffer), NULL)) {
              printf("[Client_sock: %d] [Failed to read filename
buffer] [errno: %d] [strerror: %s]\n", client_sock, errno,
strerror(errno));
```

```
            return;
        }

                                if(memchr(filename_buffer,    '\0',
ARRAY_SIZE(filename_buffer)) == NULL) {
             printf("[Client_sock: %d] [Error filename not valid]\
n", client_sock);
            const bool is_file_size_ok = false;
               if(not checked_write(client_sock, &is_file_size_ok,
sizeof(is_file_size_ok), NULL)) {
                    printf("[Client_sock: %d] [Error filename not
valid]  [errno:  %d]  [strerror:  %s]\n",  client_sock,  errno,
strerror(errno));
            }
            return;
        }
        const size_t dir_path_strlen = strlen(dir_path);
        const size_t filename_strlen = strlen(filename_buffer);
          const size_t buffer_byte_count = dir_path_strlen + 1 +
filename_strlen + 1;
        buffer = malloc(buffer_byte_count);
           snprintf(buffer, buffer_byte_count, "%s/%s", dir_path,
filename_buffer);
    }
    const int fd = open(buffer, O_RDONLY);
    if(fd == -1) {
          printf("[Client_sock: %d] [Error open file: %s] [errno:
%d]   [strerror:   %s]\n",    client_sock,    buffer,    errno,
strerror(errno));
        const bool is_file_size_ok = false;
             if(not  checked_write(client_sock,  &is_file_size_ok,
sizeof(is_file_size_ok), NULL)) {
              printf("[Client_sock: %d] [Failed to inform failure
file  size  not  ok]  [errno:  %d]  [strerror:  %s]\n",  client_sock,
errno, strerror(errno));
        }
    } else {
```

```
        with_file_open(fd, client_sock, buffer);
        if(not checked_close(fd)) {
                printf("[Client_sock: %d] [Failed to close file] [errno: %d] [strerror: %s]\n", client_sock, errno, strerror(errno));
        }
    }
    free(buffer);
}
```

```
// ./lab_3/pool_server.c

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <signal.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <errno.h>
#include <err.h>
#include <stdatomic.h>
#include "iterative_server_utils_two.h"

typedef struct {
    IterativeServerConfig config;
    int32_t max_children;
} ParallelServerConfig;

static void parallel_server_print_config(const ParallelServerConfig *config) {
    iterative_server_print_config(&config->config);
    printf("\tMaximum Children: %d\n", config->max_children);
}
```

```
static ParallelServerConfig handle_cmd_args(const int argc, const
char **argv) {
    if (argc != 5) {
                printf("Usage: %s <server_address> <server_port>
<directory_path> <max_children>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const ParallelServerConfig config = {
        .config.address = argv[1],
        .config.port = (uint16_t)atoi(argv[2]),
        .config.dir_path = argv[3],
        .max_children = atoi(argv[4])
    };
    assert(config.max_children > 0);
    parallel_server_print_config(&config);
    return config;
}

static void wait_finish_child_processes(void) {
    while (true) {
        const pid_t pid = waitpid(-1, NULL, WNOHANG);
        switch (pid) {
            case -1: {
                if (errno != ECHILD) {
                    printf("[Failed waitpid]");
                }
                return;
            }
            case 0: {
                return;
            }
            default: {
                printf("[Process %jd exited]\n", (intmax_t)pid);
            }
        }
```

```
    }
}

static void handle_sigchld(const int value __attribute__((unused))) {
    wait_finish_child_processes();
}

static void socketfd_valid(const ParallelServerConfig *config, const int socketfd) {
    {
        const struct sockaddr_in srv_sin4 = {
            .sin_family  = AF_INET,
            .sin_port = htons(config->config.port),
            .sin_addr.s_addr = inet_addr(config->config.address)
        };
              ASSERT_POSIX(bind(socketfd, (const struct sockaddr *)&srv_sin4, sizeof(srv_sin4)));
    }
    ASSERT_POSIX(listen(socketfd, MAX_BACKLOG));
            printf("[Server listening on %s:%d]\n", config->config.address, config->config.port);

    sigset_t sigcld_unblock_mask;
    sigprocmask(SIG_BLOCK, NULL, &sigcld_unblock_mask);
    sigdelset(&sigcld_unblock_mask, SIGCLD);

    sigset_t sigcld_block_mask;
    sigprocmask(SIG_BLOCK, NULL, &sigcld_block_mask);
    sigaddset(&sigcld_block_mask, SIGCHLD);

    sigprocmask(SIG_BLOCK, &sigcld_block_mask, NULL);

    for(uint32_t i = 0; i < (uint32_t)config->max_children; ++i) {
        const pid_t pid = fork();
        ASSERT_POSIX(pid);
        if(pid == 0) {
```

```
                    iterative_server_main_loop(socketfd, config->config.dir_path);
            return;
        }
    }
    iterative_server_main_loop(socketfd, config->config.dir_path);
    wait_finish_child_processes();
}

int main(const int argc, const char *argv[]) {
    {
        struct sigaction sa;
        ASSERT_POSIX(sigemptyset(&sa.sa_mask));
        sa.sa_flags = 0;

        sa.sa_handler = handle_sigint;
        ASSERT_POSIX(sigaction(SIGINT, &sa, NULL));

        sa.sa_handler = handle_sigchld;
        ASSERT_POSIX(sigaction(SIGCLD, &sa, NULL));
    }
       const ParallelServerConfig config = handle_cmd_args(argc, argv);
          const int listenfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    ASSERT_POSIX(listenfd);
    socketfd_valid(&config, listenfd);
    assert(checked_close(listenfd));
    return EXIT_SUCCESS;
}


// ./lab_3/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>


#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) != -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;
}

enum { PROTOCOL_VERSION = 17 };
typedef char filename_buff_t[255];
```