****

Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інтегрованих інформаційних систем

Лабораторна робота №3  
**Мережеве програмування у середовищі UNIX**

Тема: «Багатопроцесний ітеративний TCP клієнт-сервер*»*

|  |  |
| --- | --- |
| Виконав:  Студент групи ІА-12  Оверчук Дмитро Максимович | Перевірив:  Сімоненко А.В. |

Київ 2025

## Завдання на роботу

### 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 ди- ректорії, шляхове ім’я якої вказано в аргументі його командного рядка, та відправляє клі- єнту 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 збі- гаються, 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 імені файлу 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’єднан- ня. Розмір файлу не перевищує значення (2^64 - 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 го- товність 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 ви- значається 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 поча- ти 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 кі- лькості.
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.

### Tips:

* 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).

## Код програми

[protocol.h](https://github.com/dymytryke/Netw.UNIX/blob/lab3/3/src/protocol.h)

#ifndef PROTOCOL\_H

#define PROTOCOL\_H

#include <stdint.h>

**//** Версія протоколу

#define PROTOCOL\_VERSION 1

**//** Максимальна довжина імені файлу **(**без нульового символу**)**

#define MAX\_FILENAME\_LENGTH 255

**//** Коди помилок

#define ERR\_PROTOCOL\_MISMATCH 1

#define ERR\_INVALID\_FILENAME 2

#define ERR\_FILE\_NOT\_FOUND 3

#define ERR\_FILE\_ACCESS 4

#define ERR\_INTERNAL 5

**//** Типи повідомлень

#define MSG\_FILE\_REQUEST 1

#define MSG\_FILE\_RESPONSE 2

#define MSG\_READY\_TO\_RECEIVE 3

#define MSG\_REFUSE\_TO\_RECEIVE 4

**//** Розмір блоку при передачі файлу **(**64КБ**)**

#define DEFAULT\_CHUNK\_SIZE (64 \* 1024)

**//** Заголовок запиту від клієнта до сервера

typedef struct **{**

uint8\_t protocol\_version**;**

uint8\_t message\_type**;**

uint16\_t filename\_length**;** **//** Довжина імені файлу

char filename**[**MAX\_FILENAME\_LENGTH **+** 1**];** **//** **+**1 для термінатора '\0'

**}** FileRequestHeader**;**

**//** Заголовок відповіді від сервера до клієнта

typedef struct **{**

uint8\_t protocol\_version**;**

uint8\_t message\_type**;**

uint8\_t status**;** **//** 0 **=** успіх**,** інакше код помилки

uint64\_t file\_size**;** **//** Розмір файлу у байтах

**}** FileResponseHeader**;**

**//** Заголовок відповіді клієнта серверу

typedef struct **{**

uint8\_t message\_type**;** **//** MSG\_READY\_TO\_RECEIVE або MSG\_REFUSE\_TO\_RECEIVE

**}** ClientResponseHeader**;**

#endif // PROTOCOL\_H

[utils.h](https://github.com/dymytryke/Netw.UNIX/blob/lab3/3/src/utils.h)

#ifndef UTILS\_H

#define UTILS\_H

#include <string.h>

**//** Function to validate a filename **(**no path separators **or** invalid characters**)**

static inline **int** validate\_filename**(**const char **\***filename**)** **{**

**//** Check **for** null **or** empty filename

**if** **(!**filename **||** **!\***filename**)** **{**

**return** 0**;**

**}**

**//** Check **for** path separators

**if** **(**strchr**(**filename**,** '/'**)** **!=** NULL**)** **{**

**return** 0**;**

**}**

**//** Check **for** "." **or** ".." which could be used to navigate directory structure

**if** **(**strcmp**(**filename**,** "."**)** **==** 0 **||** strcmp**(**filename**,** ".."**)** **==** 0**)** **{**

**return** 0**;**

**}**

**//** Basic check **for** valid filename characters

**for** **(**const char **\***p **=** filename**;** **\***p**;** p**++)** **{**

**if** **(\***p **<=** 31 **||** **\***p **==** 127**)** **{** **//** Control characters

**return** 0**;**

**}**

**//** This **is** a simplified check **-** actual filesystem restrictions may vary

**if** **(**strchr**(**"<>:\"|?\*\\"**,** **\***p**)** **!=** NULL**)** **{**

**return** 0**;**

**}**

**}**

**return** 1**;**

**}**

#endif // UTILS\_H

[client.c](https://github.com/dymytryke/Netw.UNIX/blob/lab3/3/src/client.c)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <stdint.h>

#include <errno.h>

#include <fcntl.h>

#include "protocol.h"

**//** Function to **print** error messages

void print\_error**(**const char **\***message**)** **{**

perror**(**message**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Function to **print** error code description

void print\_error\_code**(int** code**)** **{**

printf**(**"Error: "**);**

switch **(**code**)** **{**

case ERR\_PROTOCOL\_MISMATCH**:**

printf**(**"Protocol version mismatch\n"**);**

**break;**

case ERR\_INVALID\_FILENAME**:**

printf**(**"Invalid filename\n"**);**

**break;**

case ERR\_FILE\_NOT\_FOUND**:**

printf**(**"File not found\n"**);**

**break;**

case ERR\_FILE\_ACCESS**:**

printf**(**"Access to file denied\n"**);**

**break;**

case ERR\_INTERNAL**:**

printf**(**"Internal server error\n"**);**

**break;**

default**:**

printf**(**"Unknown error (code %d)\n"**,** code**);**

**}**

**}**

**//** Main function

**int** main**(int** argc**,** char **\***argv**[])** **{**

**if** **(**argc **!=** 5**)** **{**

fprintf**(**stderr**,** "Usage: %s <server\_address> <server\_port> <filename> <max\_file\_size>\n"**,** argv**[**0**]);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Parse command line arguments

const char **\***server\_address **=** argv**[**1**];**

**int** server\_port **=** atoi**(**argv**[**2**]);**

const char **\***filename **=** argv**[**3**];**

uint64\_t max\_file\_size **=** strtoull**(**argv**[**4**],** NULL**,** 10**);**

**//** Validate filename length

size\_t filename\_len **=** strlen**(**filename**);**

**if** **(**filename\_len **==** 0 **||** filename\_len **>** MAX\_FILENAME\_LENGTH**)** **{**

fprintf**(**stderr**,** "Error: Filename must be between 1 and %d characters\n"**,** MAX\_FILENAME\_LENGTH**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Create socket

**int** sock **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

**if** **(**sock **<** 0**)** **{**

print\_error**(**"Socket creation failed"**);**

**}**

**//** Connect to server

struct sockaddr\_in server\_addr**;**

memset**(&**server\_addr**,** 0**,** sizeof**(**server\_addr**));**

server\_addr**.***sin\_family* **=** AF\_INET**;**

server\_addr**.***sin\_port* **=** htons**(**server\_port**);**

**if** **(**inet\_pton**(**AF\_INET**,** server\_address**,** **&**server\_addr**.***sin\_addr***)** **<=** 0**)** **{**

fprintf**(**stderr**,** "Invalid address/ Address not supported\n"**);**

close**(**sock**);**

**exit(**EXIT\_FAILURE**);**

**}**

printf**(**"Connecting to server at %s:%d\n"**,** server\_address**,** server\_port**);**

**if** **(**connect**(**sock**,** **(**struct sockaddr **\*)&**server\_addr**,** sizeof**(**server\_addr**))** **<** 0**)** **{**

print\_error**(**"Connection failed"**);**

**}**

printf**(**"Connected to server\n"**);**

**//** Prepare file request

FileRequestHeader request**;**

memset**(&**request**,** 0**,** sizeof**(**request**));**

request**.***protocol\_version* **=** PROTOCOL\_VERSION**;**

request**.***message\_type* **=** MSG\_FILE\_REQUEST**;**

request**.***filename\_length* **=** filename\_len**;**

strncpy**(**request**.***filename***,** filename**,** MAX\_FILENAME\_LENGTH**);**

printf**(**"Sending file request for: %s\n"**,** filename**);**

**//** Send file request

**if** **(**send**(**sock**,** **&**request**,** sizeof**(**request**),** 0**)** **!=** sizeof**(**request**))** **{**

print\_error**(**"Failed to send file request"**);**

**}**

**//** Receive server response

FileResponseHeader response**;**

ssize\_t bytes\_received **=** recv**(**sock**,** **&**response**,** sizeof**(**response**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**response**))** **{**

print\_error**(**"Failed to receive response header"**);**

**}**

printf**(**"Received response: protocol version %d, message type %d, status %d, file size %lu\n"**,**

response**.***protocol\_version***,** response**.***message\_type***,** response**.***status***,** response**.***file\_size***);**

**//** Check protocol version

**if** **(**response**.***protocol\_version* **!=** PROTOCOL\_VERSION**)** **{**

fprintf**(**stderr**,** "Error: Protocol version mismatch\n"**);**

close**(**sock**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Check **for** error

**if** **(**response**.***status* **!=** 0**)** **{**

print\_error\_code**(**response**.***status***);**

close**(**sock**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Check file size against maximum

ClientResponseHeader client\_response**;**

**if** **(**response**.***file\_size* **>** max\_file\_size**)** **{**

printf**(**"File size (%lu bytes) exceeds maximum allowed size (%lu bytes)\n"**,**

response**.***file\_size***,** max\_file\_size**);**

client\_response**.***message\_type* **=** MSG\_REFUSE\_TO\_RECEIVE**;**

send**(**sock**,** **&**client\_response**,** sizeof**(**client\_response**),** 0**);**

close**(**sock**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Send ready to receive

client\_response**.***message\_type* **=** MSG\_READY\_TO\_RECEIVE**;**

printf**(**"Sending ready to receive message\n"**);**

**if** **(**send**(**sock**,** **&**client\_response**,** sizeof**(**client\_response**),** 0**)** **!=** sizeof**(**client\_response**))** **{**

print\_error**(**"Failed to send ready message"**);**

**}**

**//** Create output file

**int** file\_fd **=** **open(**filename**,** O\_WRONLY **|** O\_CREAT **|** O\_TRUNC**,** 0644**);**

**if** **(**file\_fd **<** 0**)** **{**

print\_error**(**"Failed to create output file"**);**

**}**

**//** Receive file content

printf**(**"Receiving file content (%lu bytes)...\n"**,** response**.***file\_size***);**

uint64\_t total\_received **=** 0**;**

char buffer**[**4096**];**

**while** **(**total\_received **<** response**.***file\_size***)** **{**

bytes\_received **=** recv**(**sock**,** buffer**,** sizeof**(**buffer**),** 0**);**

**if** **(**bytes\_received **<=** 0**)** **{**

**if** **(**bytes\_received **==** 0**)** **{**

printf**(**"Connection closed by server\n"**);**

**break;**

**}** **else** **{**

print\_error**(**"Error receiving file content"**);**

**}**

**}**

**if** **(**write**(**file\_fd**,** buffer**,** bytes\_received**)** **!=** bytes\_received**)** **{**

print\_error**(**"Failed to write to output file"**);**

**}**

total\_received **+=** bytes\_received**;**

printf**(**"\rReceived: %lu/%lu bytes (%.1f%%)"**,**

total\_received**,** response**.***file\_size***,**

**(float)**total\_received **/** response**.***file\_size* **\*** 100**);**

fflush**(**stdout**);**

**}**

printf**(**"\nFile transfer complete\n"**);**

**//** Close file **and** socket

close**(**file\_fd**);**

close**(**sock**);**

**return** 0**;**

**}**

[iterative\_server.c](https://github.com/dymytryke/Netw.UNIX/blob/lab3/3/src/iterative_server.c)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <stdint.h>

#include <errno.h>

#include <ctype.h>

#include <dirent.h>

#include <limits.h>

#include "protocol.h"

#include "utils.h"

**//** Function to **print** error messages

void print\_error**(**const char **\***message**)** **{**

perror**(**message**);**

**}**

**//** Function to handle a client connection

void handle\_client**(int** client\_sock**,** const char **\***directory**)** **{**

**//** Receive file request

FileRequestHeader request**;**

ssize\_t bytes\_received **=** recv**(**client\_sock**,** **&**request**,** sizeof**(**request**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**request**))** **{**

printf**(**"Error: Failed to receive file request (received %zd bytes, expected %zu)\n"**,**

bytes\_received**,** sizeof**(**request**));**

close**(**client\_sock**);**

**return;**

**}**

**//** Prepare response

FileResponseHeader response**;**

memset**(&**response**,** 0**,** sizeof**(**response**));**

response**.***protocol\_version* **=** PROTOCOL\_VERSION**;**

response**.***message\_type* **=** MSG\_FILE\_RESPONSE**;**

**//** Check protocol version

**if** **(**request**.***protocol\_version* **!=** PROTOCOL\_VERSION**)** **{**

printf**(**"Error: Protocol version mismatch (client: %d, server: %d)\n"**,**

request**.***protocol\_version***,** PROTOCOL\_VERSION**);**

response**.***status* **=** ERR\_PROTOCOL\_MISMATCH**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Null**-**terminate filename to be safe

request**.***filename***[**request**.***filename\_length***]** **=** '\0'**;**

printf**(**"Received file request for: %s\n"**,** request**.***filename***);**

**//** Validate filename

**if** **(!**validate\_filename**(**request**.***filename***))** **{**

printf**(**"Error: Invalid filename: %s\n"**,** request**.***filename***);**

response**.***status* **=** ERR\_INVALID\_FILENAME**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Build full path

char filepath**[**PATH\_MAX**];**

snprintf**(**filepath**,** sizeof**(**filepath**),** "%s/%s"**,** directory**,** request**.***filename***);**

**//** Check **if** the file exists **and** **is** a regular file

struct stat file\_stat**;**

**if** **(**stat**(**filepath**,** **&**file\_stat**)** **!=** 0**)** **{**

printf**(**"Error: File not found: %s\n"**,** filepath**);**

response**.***status* **=** ERR\_FILE\_NOT\_FOUND**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**if** **(!**S\_ISREG**(**file\_stat**.***st\_mode***))** **{**

printf**(**"Error: Not a regular file: %s\n"**,** filepath**);**

response**.***status* **=** ERR\_FILE\_NOT\_FOUND**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Get file size

uint64\_t file\_size **=** file\_stat**.***st\_size***;**

**//** Prepare success response

response**.***status* **=** 0**;** **//** Success

response**.***file\_size* **=** file\_size**;**

printf**(**"Sending file response: status=%d, file\_size=%lu\n"**,**

response**.***status***,** response**.***file\_size***);**

**//** Send response

**if** **(**send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**)** **!=** sizeof**(**response**))** **{**

print\_error**(**"Failed to send file response"**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Receive client's decision

ClientResponseHeader client\_response**;**

bytes\_received **=** recv**(**client\_sock**,** **&**client\_response**,** sizeof**(**client\_response**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**client\_response**))** **{**

printf**(**"Error: Failed to receive client response\n"**);**

close**(**client\_sock**);**

**return;**

**}**

**if** **(**client\_response**.***message\_type* **==** MSG\_REFUSE\_TO\_RECEIVE**)** **{**

printf**(**"Client refused to receive the file (too large)\n"**);**

close**(**client\_sock**);**

**return;**

**}**

**if** **(**client\_response**.***message\_type* **!=** MSG\_READY\_TO\_RECEIVE**)** **{**

printf**(**"Error: Unexpected client response message type: %d\n"**,** client\_response**.***message\_type***);**

close**(**client\_sock**);**

**return;**

**}**

printf**(**"Client is ready to receive the file\n"**);**

**//** Open the file

**int** file\_fd **=** **open(**filepath**,** O\_RDONLY**);**

**if** **(**file\_fd **<** 0**)** **{**

print\_error**(**"Failed to open file"**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Send file content **in** chunks

char buffer**[**DEFAULT\_CHUNK\_SIZE**];**

ssize\_t bytes\_read**;**

uint64\_t total\_sent **=** 0**;**

printf**(**"Sending file content (%lu bytes)...\n"**,** file\_size**);**

**while** **((**bytes\_read **=** read**(**file\_fd**,** buffer**,** sizeof**(**buffer**)))** **>** 0**)** **{**

ssize\_t bytes\_sent **=** send**(**client\_sock**,** buffer**,** bytes\_read**,** 0**);**

**if** **(**bytes\_sent **!=** bytes\_read**)** **{**

print\_error**(**"Failed to send file content"**);**

close**(**file\_fd**);**

close**(**client\_sock**);**

**return;**

**}**

total\_sent **+=** bytes\_sent**;**

printf**(**"\rSent: %lu/%lu bytes (%.1f%%)"**,**

total\_sent**,** file\_size**,**

**(float)**total\_sent **/** file\_size **\*** 100**);**

fflush**(**stdout**);**

**}**

printf**(**"\nFile transfer complete\n"**);**

**//** Close file **and** client connection

close**(**file\_fd**);**

close**(**client\_sock**);**

**}**

**int** main**(int** argc**,** char **\***argv**[])** **{**

**if** **(**argc **!=** 4**)** **{**

fprintf**(**stderr**,** "Usage: %s <server\_address> <server\_port> <directory>\n"**,** argv**[**0**]);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Parse command line arguments

const char **\***server\_address **=** argv**[**1**];**

**int** server\_port **=** atoi**(**argv**[**2**]);**

const char **\***directory **=** argv**[**3**];**

**//** Check **if** directory exists

DIR **\*dir** **=** opendir**(**directory**);**

**if** **(!dir)** **{**

fprintf**(**stderr**,** "Error: Directory '%s' does not exist or is not accessible\n"**,** directory**);**

**exit(**EXIT\_FAILURE**);**

**}**

closedir**(dir);**

**//** Create socket

**int** server\_sock **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

**if** **(**server\_sock **<** 0**)** **{**

perror**(**"Socket creation failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Set socket option to reuse address

**int** opt **=** 1**;**

**if** **(**setsockopt**(**server\_sock**,** SOL\_SOCKET**,** SO\_REUSEADDR**,** **&**opt**,** sizeof**(**opt**))** **<** 0**)** **{**

perror**(**"Setsockopt failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Bind socket to specified address **and** port

struct sockaddr\_in server\_addr**;**

memset**(&**server\_addr**,** 0**,** sizeof**(**server\_addr**));**

server\_addr**.***sin\_family* **=** AF\_INET**;**

server\_addr**.***sin\_port* **=** htons**(**server\_port**);**

**if** **(**inet\_pton**(**AF\_INET**,** server\_address**,** **&**server\_addr**.***sin\_addr***)** **<=** 0**)** **{**

fprintf**(**stderr**,** "Invalid address/ Address not supported\n"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**if** **(**bind**(**server\_sock**,** **(**struct sockaddr **\*)&**server\_addr**,** sizeof**(**server\_addr**))** **<** 0**)** **{**

perror**(**"Bind failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Listen **for** connections

**if** **(**listen**(**server\_sock**,** 10**)** **<** 0**)** **{**

perror**(**"Listen failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

printf**(**"Iterative server started at %s:%d, serving files from '%s'\n"**,**

server\_address**,** server\_port**,** directory**);**

**//** Main server loop

**while** **(**1**)** **{**

struct sockaddr\_in client\_addr**;**

socklen\_t client\_len **=** sizeof**(**client\_addr**);**

printf**(**"Waiting for connections...\n"**);**

**//** Accept a new connection

**int** client\_sock **=** accept**(**server\_sock**,** **(**struct sockaddr **\*)&**client\_addr**,** **&**client\_len**);**

**if** **(**client\_sock **<** 0**)** **{**

print\_error**(**"Accept failed"**);**

**continue;** **//** Continue accepting connections

**}**

**//** Get client information

char client\_ip**[**INET\_ADDRSTRLEN**];**

inet\_ntop**(**AF\_INET**,** **&**client\_addr**.***sin\_addr***,** client\_ip**,** INET\_ADDRSTRLEN**);**

**int** client\_port **=** ntohs**(**client\_addr**.***sin\_port***);**

printf**(**"Accepted connection from %s:%d\n"**,** client\_ip**,** client\_port**);**

**//** Handle client **in** the same process **(**iterative server**)**

handle\_client**(**client\_sock**,** directory**);**

printf**(**"Client connection handled and closed\n"**);**

**}**

**//** Close server socket **(**never reached **in** this example**)**

close**(**server\_sock**);**

**return** 0**;**

**}**

[parallel\_server.c](https://github.com/dymytryke/Netw.UNIX/blob/lab3/3/src/parallel_server.c)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <stdint.h>

#include <errno.h>

#include <ctype.h>

#include <dirent.h>

#include <sys/wait.h>

#include <signal.h>

#include "protocol.h"

#include "utils.h"

**//** Global variables to track child processes

volatile **int** active\_children **=** 0**;**

**int** max\_children **=** 0**;**

**//** Function to **print** error messages

void print\_error**(**const char **\***message**)** **{**

perror**(**message**);**

**}**

**//** Signal handler **for** child process termination

void sigchld\_handler**(int** sig**)** **{**

**int** saved\_errno **=** errno**;**

**while** **(**waitpid**(-**1**,** NULL**,** WNOHANG**)** **>** 0**)** **{**

active\_children**--;**

**}**

errno **=** saved\_errno**;**

**}**

**//** Function to handle a client connection

void handle\_client**(int** client\_sock**,** const char **\***directory**)** **{**

**//** Receive file request

FileRequestHeader request**;**

ssize\_t bytes\_received **=** recv**(**client\_sock**,** **&**request**,** sizeof**(**request**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**request**))** **{**

printf**(**"Error: Failed to receive file request (received %zd bytes, expected %zu)\n"**,**

bytes\_received**,** sizeof**(**request**));**

close**(**client\_sock**);**

**return;**

**}**

**//** Prepare response

FileResponseHeader response**;**

memset**(&**response**,** 0**,** sizeof**(**response**));**

response**.***protocol\_version* **=** PROTOCOL\_VERSION**;**

response**.***message\_type* **=** MSG\_FILE\_RESPONSE**;**

**//** Check protocol version

**if** **(**request**.***protocol\_version* **!=** PROTOCOL\_VERSION**)** **{**

printf**(**"Error: Protocol version mismatch (client: %d, server: %d)\n"**,**

request**.***protocol\_version***,** PROTOCOL\_VERSION**);**

response**.***status* **=** ERR\_PROTOCOL\_MISMATCH**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Null**-**terminate filename to be safe

request**.***filename***[**request**.***filename\_length***]** **=** '\0'**;**

printf**(**"[PID %d] Received file request for: %s\n"**,** getpid**(),** request**.***filename***);**

**//** Validate filename

**if** **(!**validate\_filename**(**request**.***filename***))** **{**

printf**(**"[PID %d] Error: Invalid filename: %s\n"**,** getpid**(),** request**.***filename***);**

response**.***status* **=** ERR\_INVALID\_FILENAME**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Build full path

char filepath**[**PATH\_MAX**];**

snprintf**(**filepath**,** sizeof**(**filepath**),** "%s/%s"**,** directory**,** request**.***filename***);**

**//** Check **if** the file exists **and** **is** a regular file

struct stat file\_stat**;**

**if** **(**stat**(**filepath**,** **&**file\_stat**)** **!=** 0**)** **{**

printf**(**"[PID %d] Error: File not found: %s\n"**,** getpid**(),** filepath**);**

response**.***status* **=** ERR\_FILE\_NOT\_FOUND**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**if** **(!**S\_ISREG**(**file\_stat**.***st\_mode***))** **{**

printf**(**"[PID %d] Error: Not a regular file: %s\n"**,** getpid**(),** filepath**);**

response**.***status* **=** ERR\_FILE\_NOT\_FOUND**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Get file size

uint64\_t file\_size **=** file\_stat**.***st\_size***;**

**//** Prepare success response

response**.***status* **=** 0**;** **//** Success

response**.***file\_size* **=** file\_size**;**

printf**(**"[PID %d] Sending file response: status=%d, file\_size=%lu\n"**,**

getpid**(),** response**.***status***,** response**.***file\_size***);**

**//** Send response

**if** **(**send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**)** **!=** sizeof**(**response**))** **{**

print\_error**(**"Failed to send file response"**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Receive client's decision

ClientResponseHeader client\_response**;**

bytes\_received **=** recv**(**client\_sock**,** **&**client\_response**,** sizeof**(**client\_response**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**client\_response**))** **{**

printf**(**"[PID %d] Error: Failed to receive client response\n"**,** getpid**());**

close**(**client\_sock**);**

**return;**

**}**

**if** **(**client\_response**.***message\_type* **==** MSG\_REFUSE\_TO\_RECEIVE**)** **{**

printf**(**"[PID %d] Client refused to receive the file (too large)\n"**,** getpid**());**

close**(**client\_sock**);**

**return;**

**}**

**if** **(**client\_response**.***message\_type* **!=** MSG\_READY\_TO\_RECEIVE**)** **{**

printf**(**"[PID %d] Error: Unexpected client response message type: %d\n"**,**

getpid**(),** client\_response**.***message\_type***);**

close**(**client\_sock**);**

**return;**

**}**

printf**(**"[PID %d] Client is ready to receive the file\n"**,** getpid**());**

**//** Open the file

**int** file\_fd **=** **open(**filepath**,** O\_RDONLY**);**

**if** **(**file\_fd **<** 0**)** **{**

print\_error**(**"Failed to open file"**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Send file content **in** chunks

char buffer**[**DEFAULT\_CHUNK\_SIZE**];**

ssize\_t bytes\_read**;**

uint64\_t total\_sent **=** 0**;**

printf**(**"[PID %d] Sending file content (%lu bytes)...\n"**,** getpid**(),** file\_size**);**

**while** **((**bytes\_read **=** read**(**file\_fd**,** buffer**,** sizeof**(**buffer**)))** **>** 0**)** **{**

ssize\_t bytes\_sent **=** send**(**client\_sock**,** buffer**,** bytes\_read**,** 0**);**

**if** **(**bytes\_sent **!=** bytes\_read**)** **{**

print\_error**(**"Failed to send file content"**);**

close**(**file\_fd**);**

close**(**client\_sock**);**

**return;**

**}**

total\_sent **+=** bytes\_sent**;**

printf**(**"[PID %d] \rSent: %lu/%lu bytes (%.1f%%)"**,**

getpid**(),** total\_sent**,** file\_size**,**

**(float)**total\_sent **/** file\_size **\*** 100**);**

fflush**(**stdout**);**

**}**

printf**(**"\n[PID %d] File transfer complete\n"**,** getpid**());**

**//** Close file **and** client connection

close**(**file\_fd**);**

close**(**client\_sock**);**

**}**

**int** main**(int** argc**,** char **\***argv**[])** **{**

**if** **(**argc **!=** 5**)** **{**

fprintf**(**stderr**,** "Usage: %s <server\_address> <server\_port> <directory> <max\_children>\n"**,** argv**[**0**]);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Parse command line arguments

const char **\***server\_address **=** argv**[**1**];**

**int** server\_port **=** atoi**(**argv**[**2**]);**

const char **\***directory **=** argv**[**3**];**

max\_children **=** atoi**(**argv**[**4**]);**

**if** **(**max\_children **<=** 0**)** **{**

fprintf**(**stderr**,** "Error: max\_children must be a positive integer\n"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Check **if** directory exists

DIR **\*dir** **=** opendir**(**directory**);**

**if** **(!dir)** **{**

fprintf**(**stderr**,** "Error: Directory '%s' does not exist or is not accessible\n"**,** directory**);**

**exit(**EXIT\_FAILURE**);**

**}**

closedir**(dir);**

**//** Set up signal handler **for** child process termination

struct sigaction sa**;**

sa**.***sa\_handler* **=** sigchld\_handler**;**

sigemptyset**(&**sa**.***sa\_mask***);**

sa**.***sa\_flags* **=** SA\_RESTART**;**

**if** **(**sigaction**(**SIGCHLD**,** **&**sa**,** NULL**)** **==** **-**1**)** **{**

perror**(**"sigaction"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Create socket

**int** server\_sock **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

**if** **(**server\_sock **<** 0**)** **{**

perror**(**"Socket creation failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Set socket option to reuse address

**int** opt **=** 1**;**

**if** **(**setsockopt**(**server\_sock**,** SOL\_SOCKET**,** SO\_REUSEADDR**,** **&**opt**,** sizeof**(**opt**))** **<** 0**)** **{**

perror**(**"Setsockopt failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Bind socket to specified address **and** port

struct sockaddr\_in server\_addr**;**

memset**(&**server\_addr**,** 0**,** sizeof**(**server\_addr**));**

server\_addr**.***sin\_family* **=** AF\_INET**;**

server\_addr**.***sin\_port* **=** htons**(**server\_port**);**

**if** **(**inet\_pton**(**AF\_INET**,** server\_address**,** **&**server\_addr**.***sin\_addr***)** **<=** 0**)** **{**

fprintf**(**stderr**,** "Invalid address/ Address not supported\n"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**if** **(**bind**(**server\_sock**,** **(**struct sockaddr **\*)&**server\_addr**,** sizeof**(**server\_addr**))** **<** 0**)** **{**

perror**(**"Bind failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Listen **for** connections

**if** **(**listen**(**server\_sock**,** 10**)** **<** 0**)** **{**

perror**(**"Listen failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

printf**(**"Parallel server started at %s:%d, serving files from '%s', max children: %d\n"**,**

server\_address**,** server\_port**,** directory**,** max\_children**);**

**//** Main server loop

**while** **(**1**)** **{**

**//** If we've reached the maximum number of children, wait

**while** **(**active\_children **>=** max\_children**)** **{**

printf**(**"Max children reached (%d), waiting for a child to exit...\n"**,** max\_children**);**

sleep**(**1**);** **//** Wait a bit to avoid busy**-**waiting

**}**

struct sockaddr\_in client\_addr**;**

socklen\_t client\_len **=** sizeof**(**client\_addr**);**

printf**(**"Waiting for connections (active children: %d/%d)...\n"**,**

active\_children**,** max\_children**);**

**//** Accept a new connection

**int** client\_sock **=** accept**(**server\_sock**,** **(**struct sockaddr **\*)&**client\_addr**,** **&**client\_len**);**

**if** **(**client\_sock **<** 0**)** **{**

**if** **(**errno **==** EINTR**)** **{**

**//** Interrupted by signal**,** **try** again

**continue;**

**}**

print\_error**(**"Accept failed"**);**

**continue;** **//** Continue accepting connections

**}**

**//** Get client information

char client\_ip**[**INET\_ADDRSTRLEN**];**

inet\_ntop**(**AF\_INET**,** **&**client\_addr**.***sin\_addr***,** client\_ip**,** INET\_ADDRSTRLEN**);**

**int** client\_port **=** ntohs**(**client\_addr**.***sin\_port***);**

printf**(**"Accepted connection from %s:%d\n"**,** client\_ip**,** client\_port**);**

**//** Fork a child process to handle the client

pid\_t pid **=** fork**();**

**if** **(**pid **<** 0**)** **{**

print\_error**(**"Fork failed"**);**

close**(**client\_sock**);**

**continue;**

**}**

**if** **(**pid **==** 0**)** **{**

**//** Child process

close**(**server\_sock**);** **//** Child doesn't need the server socket

printf**(**"[PID %d] Child process created to handle client %s:%d\n"**,**

getpid**(),** client\_ip**,** client\_port**);**

handle\_client**(**client\_sock**,** directory**);**

printf**(**"[PID %d] Client connection handled and closed\n"**,** getpid**());**

**exit(**EXIT\_SUCCESS**);**

**}** **else** **{**

**//** Parent process

active\_children**++;**

printf**(**"Child process created (PID: %d), active children: %d/%d\n"**,**

pid**,** active\_children**,** max\_children**);**

close**(**client\_sock**);** **//** Parent doesn't need the client socket

**}**

**}**

**//** Close server socket **(**never reached **in** this example**)**

close**(**server\_sock**);**

**return** 0**;**

**}**

[preforked\_server.c](https://github.com/dymytryke/Netw.UNIX/blob/lab3/3/src/preforked_server.c)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <stdint.h>

#include <errno.h>

#include <ctype.h>

#include <dirent.h>

#include <sys/wait.h>

#include <signal.h>

#include "protocol.h"

#include "utils.h"

**//** Global variables **for** signal handling

volatile sig\_atomic\_t terminate **=** 0**;** **//** Flag to terminate child processes

**//** Function to **print** error messages

void print\_error**(**const char **\***message**)** **{**

perror**(**message**);**

**}**

**//** Signal handler **for** termination signals

void termination\_handler**(int** sig**)** **{**

terminate **=** 1**;**

**}**

**//** Function to handle a client connection

void handle\_client**(int** client\_sock**,** const char **\***directory**)** **{**

**//** Receive file request

FileRequestHeader request**;**

ssize\_t bytes\_received **=** recv**(**client\_sock**,** **&**request**,** sizeof**(**request**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**request**))** **{**

printf**(**"[PID %d] Error: Failed to receive file request (received %zd bytes, expected %zu)\n"**,**

getpid**(),** bytes\_received**,** sizeof**(**request**));**

close**(**client\_sock**);**

**return;**

**}**

**//** Prepare response

FileResponseHeader response**;**

memset**(&**response**,** 0**,** sizeof**(**response**));**

response**.***protocol\_version* **=** PROTOCOL\_VERSION**;**

response**.***message\_type* **=** MSG\_FILE\_RESPONSE**;**

**//** Check protocol version

**if** **(**request**.***protocol\_version* **!=** PROTOCOL\_VERSION**)** **{**

printf**(**"[PID %d] Error: Protocol version mismatch (client: %d, server: %d)\n"**,**

getpid**(),** request**.***protocol\_version***,** PROTOCOL\_VERSION**);**

response**.***status* **=** ERR\_PROTOCOL\_MISMATCH**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Null**-**terminate filename to be safe

request**.***filename***[**request**.***filename\_length***]** **=** '\0'**;**

printf**(**"[PID %d] Received file request for: %s\n"**,** getpid**(),** request**.***filename***);**

**//** Validate filename

**if** **(!**validate\_filename**(**request**.***filename***))** **{**

printf**(**"[PID %d] Error: Invalid filename: %s\n"**,** getpid**(),** request**.***filename***);**

response**.***status* **=** ERR\_INVALID\_FILENAME**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Build full path

char filepath**[**PATH\_MAX**];**

snprintf**(**filepath**,** sizeof**(**filepath**),** "%s/%s"**,** directory**,** request**.***filename***);**

**//** Check **if** the file exists **and** **is** a regular file

struct stat file\_stat**;**

**if** **(**stat**(**filepath**,** **&**file\_stat**)** **!=** 0**)** **{**

printf**(**"[PID %d] Error: File not found: %s\n"**,** getpid**(),** filepath**);**

response**.***status* **=** ERR\_FILE\_NOT\_FOUND**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**if** **(!**S\_ISREG**(**file\_stat**.***st\_mode***))** **{**

printf**(**"[PID %d] Error: Not a regular file: %s\n"**,** getpid**(),** filepath**);**

response**.***status* **=** ERR\_FILE\_NOT\_FOUND**;**

send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Get file size

uint64\_t file\_size **=** file\_stat**.***st\_size***;**

**//** Prepare success response

response**.***status* **=** 0**;** **//** Success

response**.***file\_size* **=** file\_size**;**

printf**(**"[PID %d] Sending file response: status=%d, file\_size=%lu\n"**,**

getpid**(),** response**.***status***,** response**.***file\_size***);**

**//** Send response

**if** **(**send**(**client\_sock**,** **&**response**,** sizeof**(**response**),** 0**)** **!=** sizeof**(**response**))** **{**

print\_error**(**"Failed to send file response"**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Receive client's decision

ClientResponseHeader client\_response**;**

bytes\_received **=** recv**(**client\_sock**,** **&**client\_response**,** sizeof**(**client\_response**),** 0**);**

**if** **(**bytes\_received **!=** sizeof**(**client\_response**))** **{**

printf**(**"[PID %d] Error: Failed to receive client response\n"**,** getpid**());**

close**(**client\_sock**);**

**return;**

**}**

**if** **(**client\_response**.***message\_type* **==** MSG\_REFUSE\_TO\_RECEIVE**)** **{**

printf**(**"[PID %d] Client refused to receive the file (too large)\n"**,** getpid**());**

close**(**client\_sock**);**

**return;**

**}**

**if** **(**client\_response**.***message\_type* **!=** MSG\_READY\_TO\_RECEIVE**)** **{**

printf**(**"[PID %d] Error: Unexpected client response message type: %d\n"**,**

getpid**(),** client\_response**.***message\_type***);**

close**(**client\_sock**);**

**return;**

**}**

printf**(**"[PID %d] Client is ready to receive the file\n"**,** getpid**());**

**//** Open the file

**int** file\_fd **=** **open(**filepath**,** O\_RDONLY**);**

**if** **(**file\_fd **<** 0**)** **{**

print\_error**(**"Failed to open file"**);**

close**(**client\_sock**);**

**return;**

**}**

**//** Send file content **in** chunks

char buffer**[**DEFAULT\_CHUNK\_SIZE**];**

ssize\_t bytes\_read**;**

uint64\_t total\_sent **=** 0**;**

printf**(**"[PID %d] Sending file content (%lu bytes)...\n"**,** getpid**(),** file\_size**);**

**while** **((**bytes\_read **=** read**(**file\_fd**,** buffer**,** sizeof**(**buffer**)))** **>** 0**)** **{**

ssize\_t bytes\_sent **=** send**(**client\_sock**,** buffer**,** bytes\_read**,** 0**);**

**if** **(**bytes\_sent **!=** bytes\_read**)** **{**

print\_error**(**"Failed to send file content"**);**

close**(**file\_fd**);**

close**(**client\_sock**);**

**return;**

**}**

total\_sent **+=** bytes\_sent**;**

printf**(**"[PID %d] \rSent: %lu/%lu bytes (%.1f%%)"**,**

getpid**(),** total\_sent**,** file\_size**,**

**(float)**total\_sent **/** file\_size **\*** 100**);**

fflush**(**stdout**);**

**}**

printf**(**"\n[PID %d] File transfer complete\n"**,** getpid**());**

**//** Close file **and** client connection

close**(**file\_fd**);**

close**(**client\_sock**);**

**}**

**//** Function **for** child process to handle client connections

void child\_process**(int** server\_sock**,** const char **\***directory**)** **{**

**//** Set up signal handler **for** termination

struct sigaction sa**;**

sa**.***sa\_handler* **=** termination\_handler**;**

sigemptyset**(&**sa**.***sa\_mask***);**

sa**.***sa\_flags* **=** 0**;**

sigaction**(**SIGTERM**,** **&**sa**,** NULL**);**

printf**(**"[PID %d] Child process started, waiting for connections\n"**,** getpid**());**

**while** **(!**terminate**)** **{**

struct sockaddr\_in client\_addr**;**

socklen\_t client\_len **=** sizeof**(**client\_addr**);**

**//** Accept a new connection

**int** client\_sock **=** accept**(**server\_sock**,** **(**struct sockaddr **\*)&**client\_addr**,** **&**client\_len**);**

**if** **(**client\_sock **<** 0**)** **{**

**if** **(**errno **==** EINTR**)** **{**

**//** Interrupted by signal**,** check **if** we need to terminate

**if** **(**terminate**)** **{**

**break;**

**}**

**continue;**

**}**

print\_error**(**"Accept failed"**);**

**continue;** **//** Continue accepting connections

**}**

**//** Get client information

char client\_ip**[**INET\_ADDRSTRLEN**];**

inet\_ntop**(**AF\_INET**,** **&**client\_addr**.***sin\_addr***,** client\_ip**,** INET\_ADDRSTRLEN**);**

**int** client\_port **=** ntohs**(**client\_addr**.***sin\_port***);**

printf**(**"[PID %d] Accepted connection from %s:%d\n"**,** getpid**(),** client\_ip**,** client\_port**);**

**//** Handle the client

handle\_client**(**client\_sock**,** directory**);**

printf**(**"[PID %d] Client connection handled and closed\n"**,** getpid**());**

**}**

printf**(**"[PID %d] Child process terminating\n"**,** getpid**());**

**exit(**EXIT\_SUCCESS**);**

**}**

**int** main**(int** argc**,** char **\***argv**[])** **{**

**if** **(**argc **!=** 5**)** **{**

fprintf**(**stderr**,** "Usage: %s <server\_address> <server\_port> <directory> <num\_children>\n"**,** argv**[**0**]);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Parse command line arguments

const char **\***server\_address **=** argv**[**1**];**

**int** server\_port **=** atoi**(**argv**[**2**]);**

const char **\***directory **=** argv**[**3**];**

**int** num\_children **=** atoi**(**argv**[**4**]);**

**if** **(**num\_children **<=** 0**)** **{**

fprintf**(**stderr**,** "Error: num\_children must be a positive integer\n"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Check **if** directory exists

DIR **\*dir** **=** opendir**(**directory**);**

**if** **(!dir)** **{**

fprintf**(**stderr**,** "Error: Directory '%s' does not exist or is not accessible\n"**,** directory**);**

**exit(**EXIT\_FAILURE**);**

**}**

closedir**(dir);**

**//** Create socket

**int** server\_sock **=** socket**(**AF\_INET**,** SOCK\_STREAM**,** 0**);**

**if** **(**server\_sock **<** 0**)** **{**

perror**(**"Socket creation failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Set socket option to reuse address

**int** opt **=** 1**;**

**if** **(**setsockopt**(**server\_sock**,** SOL\_SOCKET**,** SO\_REUSEADDR**,** **&**opt**,** sizeof**(**opt**))** **<** 0**)** **{**

perror**(**"Setsockopt failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Bind socket to specified address **and** port

struct sockaddr\_in server\_addr**;**

memset**(&**server\_addr**,** 0**,** sizeof**(**server\_addr**));**

server\_addr**.***sin\_family* **=** AF\_INET**;**

server\_addr**.***sin\_port* **=** htons**(**server\_port**);**

**if** **(**inet\_pton**(**AF\_INET**,** server\_address**,** **&**server\_addr**.***sin\_addr***)** **<=** 0**)** **{**

fprintf**(**stderr**,** "Invalid address/ Address not supported\n"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**if** **(**bind**(**server\_sock**,** **(**struct sockaddr **\*)&**server\_addr**,** sizeof**(**server\_addr**))** **<** 0**)** **{**

perror**(**"Bind failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Listen **for** connections

**if** **(**listen**(**server\_sock**,** 10**)** **<** 0**)** **{**

perror**(**"Listen failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

printf**(**"Pre-forked server started at %s:%d, serving files from '%s', creating %d child processes\n"**,**

server\_address**,** server\_port**,** directory**,** num\_children**);**

**//** Store child PIDs

pid\_t **\***child\_pids **=** malloc**(**num\_children **\*** sizeof**(**pid\_t**));**

**if** **(!**child\_pids**)** **{**

perror**(**"Memory allocation failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**//** Pre**-**fork child processes

**for** **(int** i **=** 0**;** i **<** num\_children**;** i**++)** **{**

pid\_t pid **=** fork**();**

**if** **(**pid **<** 0**)** **{**

perror**(**"Fork failed"**);**

**exit(**EXIT\_FAILURE**);**

**}**

**if** **(**pid **==** 0**)** **{**

**//** Child process

free**(**child\_pids**);** **//** Child doesn't need this array

child\_process**(**server\_sock**,** directory**);**

**//** Should **not** reach here

**exit(**EXIT\_SUCCESS**);**

**}** **else** **{**

**//** Parent process

child\_pids**[**i**]** **=** pid**;**

printf**(**"Child process created (PID: %d, %d/%d)\n"**,** pid**,** i **+** 1**,** num\_children**);**

**}**

**}**

printf**(**"All child processes created. Parent process is now waiting.\n"**);**

**//** Set up signal handler **for** parent to **pass** signals to children

struct sigaction sa**;**

sa**.***sa\_handler* **=** termination\_handler**;**

sigemptyset**(&**sa**.***sa\_mask***);**

sa**.***sa\_flags* **=** 0**;**

sigaction**(**SIGINT**,** **&**sa**,** NULL**);**

sigaction**(**SIGTERM**,** **&**sa**,** NULL**);**

**//** Wait **for** termination signal

**while** **(!**terminate**)** **{**

sleep**(**1**);**

**}**

printf**(**"Parent process received termination signal. Shutting down...\n"**);**

**//** Signal **all** children to terminate

**for** **(int** i **=** 0**;** i **<** num\_children**;** i**++)** **{**

printf**(**"Sending SIGTERM to child process %d\n"**,** child\_pids**[**i**]);**

kill**(**child\_pids**[**i**],** SIGTERM**);**

**}**

**//** Wait **for** **all** children to **exit**

printf**(**"Waiting for all child processes to exit...\n"**);**

**for** **(int** i **=** 0**;** i **<** num\_children**;** i**++)** **{**

**int** status**;**

waitpid**(**child\_pids**[**i**],** **&**status**,** 0**);**

printf**(**"Child process %d exited with status %d\n"**,** child\_pids**[**i**],**

WIFEXITED**(**status**)** **?** WEXITSTATUS**(**status**)** **:** **-**1**);**

**}**

free**(**child\_pids**);**

close**(**server\_sock**);**

printf**(**"Server shutdown complete\n"**);**

**return** 0**;**

**}**

## Опис програми

### 1. Відмінності між серверами

Кожна реалізація сервера має свої переваги та недоліки, їх ефективність залежить від характеру навантаження:

* **Ітеративний сервер** – обробляє лише одного клієнта за раз. Він простий у реалізації (відсутнє розділення на процеси), займає мінімум ресурсів і виключає накладні витрати на створення процесів. Однак при одночасних запитах інші клієнти чекають завершення поточної передачі, що значно знижує продуктивність у сценаріях з багатьма клієнтами. Такий сервер підходить для одиночних запитів або відлагодження, але погано масштабується під навантаженням.
* **Паралельний сервер з fork** – підтримує багатозадачність, створюючи окремий процес на кожного клієнта. Це дозволяє обслуговувати відразу декілька клієнтів (до заданого ліміту), ефективно використовуючи багатоядерність CPU. Паралельний підхід значно швидший для багатокористувацької роботи, оскільки, наприклад, один клієнт може отримувати великий файл, тоді як інший у той самий час вже підключений і теж отримує свій файл. Недоліком є витрати на створення і завершення процесів при кожному з’єднанні: fork копіює процес, що потребує часу і пам’яті. При великій кількості коротких запитів ці витрати можуть скласти помітну долю. Введення обмеження на максимальну кількість одночасних процесів запобігає вичерпуванню ресурсів, але якщо ліміт занадто малий, то при піковому навантаженні зайві клієнти чекатимуть, хоча й паралельно обслуговуючи групу клієнтів.
* **Префоркований сервер** – поєднує паралельність з відсутністю накладних витрат на кожне нове підключення. Наперед створений пул процесів дозволяє відразу приймати підключення, що особливо ефективно при високому і постійному навантаженні (наприклад, сервер, до якого одночасно звертаються десятки клієнтів). Цей сервер має стабільний рівень використання ресурсів: фіксована кількість процесів постійно зайнята очікуванням або обробкою, і не витрачається час на запуск/завершення процесу для кожного клієнта. У порівнянні з динамічним fork-підходом, префорк дає виграш у продуктивності при великому потоці запитів. Якщо ж клієнтські запити надходять рідко, префоркований сервер все одно тримає запущені процеси (деякі можуть простоювати), тобто може використовувати більше пам’яті, ніж паралельний сервер з fork, який у стані спокою має лише один процес. Також префоркована модель складніша в реалізації, адже потребує механізмів керування пулом (коректне завершення дітей тощо), але в даній лабораторній роботі це реалізовано через пересилання сигналів SIGTERM усім дочірнім процесам при зупинці сервера.

### 2. Протокол взаємодії

Клієнт і сервер обмінюються повідомленнями за заздалегідь визначеним протоколом прикладного рівня, який має чітко визначений формат структур даних:

* **Запит файлу (FileRequest):**
  + **protocol\_version** (1 байт) – версія протоколу (у даній реалізації використовується версія 1).
  + **message\_type** (1 байт) – код повідомлення, що позначає запит на файл (MSG\_FILE\_REQUEST).
  + **filename\_length** (2 байти) – фактична довжина назви файлу.
  + **filename** – рядок ASCII, що містить лише назву файлу (без шляхів), обмежений до 255 символів плюс термінатор.
* **Відповідь на запит (FileResponse):**
  + **protocol\_version** (1 байт) – версія протоколу, що має збігатися з версією клієнта.
  + **message\_type** (1 байт) – код повідомлення, що позначає відповідь (MSG\_FILE\_RESPONSE).
  + **status** (1 байт) – статус виконання запиту (0 – успіх; інші значення – коди помилок).
  + **file\_size** (8 байт) – розмір файлу у байтах (значущий лише при успішному запиті).
* **Відповідь клієнта (ClientResponse):**
  + **message\_type** (1 байт) – повідомлення, яке сигналізує про готовність приймати файл (MSG\_READY\_TO\_RECEIVE) або про відмову (MSG\_REFUSE\_TO\_RECEIVE).
* **Передача файлу:**  
  Після успішного обміну заголовками і підтвердження готовності клієнта, сервер передає вміст файлу у вигляді послідовних блоків байтів (розмір блоку визначено за замовчуванням, наприклад, 64 КБ). Завершення передачі відзначається закриттям TCP-з’єднання.

### 3. Обробка помилок

Система передбачає обробку декількох типових помилок:

* **Невідповідність версії протоколу:**  
  Якщо версії клієнта і сервера не співпадають, сервер відправляє повідомлення з кодом помилки і закриває з’єднання, а клієнт повідомляє про невідповідність.
* **Некоректне ім’я файлу:**  
  При виявленні заборонених символів або недопустимого формату імені (наприклад, містить шлях або є порожнім) сервер негайно повертає код помилки, що сигналізує про недопустиме ім’я файлу, і припиняє обробку запиту.
* **Файл не знайдено або недоступний:**  
  Якщо файл не існує або сервер не може отримати до нього доступ, повертається відповідний код помилки, і з’єднання закривається. У цьому випадку клієнт повідомляє про те, що файл не знайдено.
* **Відмова клієнта приймати файл:**  
  Якщо розмір файлу перевищує встановлений ліміт, клієнт відразу відправляє повідомлення про відмову приймати файл, і з’єднання завершується, що запобігає марній передачі даних.
* **Помилки мережі та системні помилки:**  
  Якщо при мережевих операціях або роботі з файловою системою виникають помилки, сервер обробляє їх для конкретного з’єднання (надсилаючи повідомлення про помилку або закриваючи сокет), але продовжує свою роботу для інших клієнтів. Таким чином, і клієнт, і сервер завжди отримують зрозумілі повідомлення про помилки і коректно завершають сесію.

## [Приклади використання програми](https://github.com/dymytryke/Netw.UNIX/blob/main/3/documents/Lab3.md" \l "%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%B8-%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8)