МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГАОУ ВО «СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ» ИНСТИТУТ ПЕРСПЕКТИВНОЙ ИНЖЕНЕРИИ КАФЕДРА ПРИКЛАДНОЙ ИНФОРМАТИКИ

**Лабораторная работа № 4**

по дисциплине

«Многопоточное программирование»

**Выполнил:**

Мухамеджанов Артур Тимурович

Студент 3 курса

группы ПИН-б-о-22-1

Направления подготовки

09.03.03 Прикладная информатика

очной формы обучения

**Работу принял**

Мартыновская А. С.

(ФИО)

**Тема:** Разработка и оптимизация многопоточных систем с использованием механизмов синхронизации в C++. Многопоточность и псевдомногопоточность

**Ход работы**:

**Часть 1**

**Задание 24**:Производственная линия

**Условие**: Разработайте имитатор производственной линии, изготавливающей винтики (widget). Винтик собирается из детали С и модуля, который, в свою очередь, состоит из деталей А и В. Для изготовления детали А требуется 1 секунда, В две секунды, С три секунды. Задержку изготовления деталей имитируйте при помощи sleep. Используйте семафоры-счетчики.

#include <iostream>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t sem\_a, sem\_b, sem\_c, sem\_module; // Семафоры для деталей и модуля

volatile bool running = true; // Флаг для завершения потоков

void \* produce\_a(void \* ) {

while (running) {

sleep(1); // Изготовление детали A

std::cout << "Produced A\n";

sem\_post( & sem\_a); // Увеличиваем счётчик деталей A

}

return nullptr;

}

void \* produce\_b(void \* ) {

while (running) {

sleep(2); // Изготовление детали B

std::cout << "Produced B\n";

sem\_post( & sem\_b); // Увеличиваем счётчик деталей B

}

return nullptr;

}

void \* produce\_c(void \* ) {

while (running) {

sleep(3); // Изготовление детали C

std::cout << "Produced C\n";

sem\_post( & sem\_c); // Увеличиваем счётчик деталей C

}

return nullptr;

}

void \* produce\_module(void \* ) {

while (running) {

sem\_wait( & sem\_a); // Ожидаем деталь A

sem\_wait( & sem\_b); // Ожидаем деталь B

std::cout << "Assembled module (A + B)\n";

sem\_post( & sem\_module); // Увеличиваем счётчик модулей

}

return nullptr;

}

int main() {

// Инициализация семафоров

sem\_init( & sem\_a, 0, 0);

sem\_init( & sem\_b, 0, 0);

sem\_init( & sem\_c, 0, 0);

sem\_init( & sem\_module, 0, 0);

// Создание потоков

pthread\_t thread\_a, thread\_b, thread\_c, thread\_module;

pthread\_create( & thread\_a, nullptr, produce\_a, nullptr);

pthread\_create( & thread\_b, nullptr, produce\_b, nullptr);

pthread\_create( & thread\_c, nullptr, produce\_c, nullptr);

pthread\_create( & thread\_module, nullptr, produce\_module, nullptr);

// Сборка винтиков (основной поток)

for (int i = 0; i < 5; ++i) { // Собираем 5 винтиков для демонстрации

sem\_wait( & sem\_module); // Ожидаем модуль

sem\_wait( & sem\_c); // Ожидаем деталь C

std::cout << "Assembled widget " << i + 1 << " (Module + C)\n";

}

// Завершение работы

running = false;

sem\_post( & sem\_a);

sem\_post( & sem\_b);

sem\_post( & sem\_c);

sem\_post( & sem\_module); // Разблокируем потоки

pthread\_join(thread\_a, nullptr);

pthread\_join(thread\_b, nullptr);

pthread\_join(thread\_c, nullptr);

pthread\_join(thread\_module, nullptr);

// Уничтожение семафоров

sem\_destroy( & sem\_a);

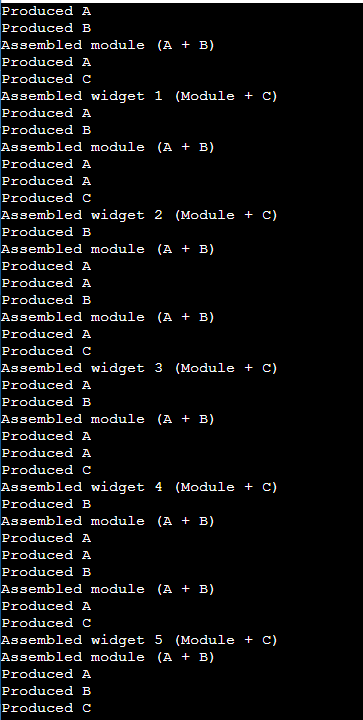
sem\_destroy( & sem\_b);

sem\_destroy( & sem\_c);

sem\_destroy( & sem\_module);

return 0;

}



**Задание 25**:Производитель-потребитель

**Условие**: Реализуйте очередь сообщений, которая может использоваться для обмена данными между двумя или большим количеством потоков. Реализация очереди должна поддерживать функции:

void mymsginit(queue \*);

void mymsqdrop(queue \*);

void mymsgdestroy(queue \*);

int mymsgput(queue \*, char \*msg);

int mymsgget(queue", char\* buf, size\_t bufsize);

Допускается реализация на С++ с заменой mymsginit и mymsgdestroy на конструктор и деструктор, а операций gеt и put на соответствующие методы.

mymsgput принимает в качестве параметра ASCIIZ строку символов, обрезает ее до 80 символов (если это необходимо) и помещает ее в очередь. Если очередь содержит более 10 записей, mymsgput блокируется. Функция возвращает количество переданных символов.

mymsgget возвращает первую запись из очереди, обрезая ее до размера пользовательского буфера (если это необходимо). В любом случае, запись извлекается из очереди полностью. Если очередь пуста, тymsgget блокируется. Функция возвращает количество прочитанных символов. mymsgdrop должна приводить к разблокированию ожидающих операций get и put. Ожидавшие вызовы и все последующие вызовы get и put должны возвращать 0.

mymsqdestroy должна вызываться после того, как будет известно, что ни один поток больше не попытается выполнять операции над очередью.

Необходимо продемонстрировать работу очереди с двумя производителями и двумя потребителями.

Для синхронизации доступа к очереди используйте семафоры-счетчики.

#include <iostream>

#include <deque>

#include <string>

#include <pthread.h>

#include <semaphore.h>

#include <cstring>

#include <unistd.h>

class MessageQueue {

private: std::deque < std::string > queue;

sem\_t sem\_slots; // Счётчик свободных слотов

sem\_t sem\_items; // Счётчик сообщений

pthread\_mutex\_t mutex; // Мьютекс для доступа к очереди

bool dropped; // Флаг сброса очереди

public: MessageQueue(): dropped(false) {

sem\_init( & sem\_slots, 0, 10); // Максимум 10 слотов

sem\_init( & sem\_items, 0, 0); // Изначально 0 сообщений

pthread\_mutex\_init( & mutex, nullptr);

}

~MessageQueue() {

mymsgdestroy();

}

int mymsgput(const char \* msg) {

if (dropped) return 0;

// Обрезаем строку до 80 символов

std::string str(msg);

if (str.size() > 80) str = str.substr(0, 80);

int len = str.size();

sem\_wait( & sem\_slots); // Ждём свободный слот

if (dropped) {

sem\_post( & sem\_slots); // Восстанавливаем слот

return 0;

}

pthread\_mutex\_lock( & mutex);

queue.push\_back(str);

pthread\_mutex\_unlock( & mutex);

sem\_post( & sem\_items); // Увеличиваем счётчик сообщений

return len;

}

int mymsgget(char \* buf, size\_t bufsize) {

if (dropped) return 0;

sem\_wait( & sem\_items); // Ждём сообщение

if (dropped) {

sem\_post( & sem\_items); // Восстанавливаем счётчик

return 0;

}

pthread\_mutex\_lock( & mutex);

std::string msg = queue.front();

queue.pop\_front();

pthread\_mutex\_unlock( & mutex);

// Обрезаем до размера буфера

size\_t copy\_len = std::min(msg.size(), bufsize - 1);

strncpy(buf, msg.c\_str(), copy\_len);

buf[copy\_len] = '\0';

sem\_post( & sem\_slots); // Освобождаем слот

return copy\_len;

}

void mymsgdrop() {

pthread\_mutex\_lock( & mutex);

dropped = true;

queue.clear();

pthread\_mutex\_unlock( & mutex);

// Разблокируем все ожидающие потоки

for (int i = 0; i < 10; ++i) {

sem\_post( & sem\_slots);

sem\_post( & sem\_items);

}

}

void mymsgdestroy() {

mymsgdrop(); // Гарантируем разблокировку

pthread\_mutex\_destroy( & mutex);

sem\_destroy( & sem\_slots);

sem\_destroy( & sem\_items);

}

};

// Потоки-производители

void \* producer(void \* arg) {

MessageQueue \* q = (MessageQueue \* ) arg;

for (int i = 1; i <= 5; ++i) {

std::string msg = "Message " + std::to\_string(i);

int len = q -> mymsgput(msg.c\_str());

if (len == 0) break;

std::cout << "Producer sent: " << msg << " (" << len << " chars)\n";

sleep(1);

}

return nullptr;

}

// Потоки-потребители

void \* consumer(void \* arg) {

MessageQueue \* q = (MessageQueue \* ) arg;

char buf[100];

while (true) {

int len = q -> mymsgget(buf, sizeof(buf));

if (len == 0) break;

std::cout << "Consumer received: " << buf << " (" << len << " chars)\n";

sleep(2);

}

return nullptr;

}

int main() {

MessageQueue q;

// Создание потоков

pthread\_t prod1, prod2, cons1, cons2;

pthread\_create( & prod1, nullptr, producer, & q);

pthread\_create( & prod2, nullptr, producer, & q);

pthread\_create( & cons1, nullptr, consumer, & q);

pthread\_create( & cons2, nullptr, consumer, & q);

// Ждём 10 секунд, затем сбрасываем очередь

sleep(10);

std::cout << "Dropping queue\n";

q.mymsgdrop();

// Ждём завершения потоков

pthread\_join(prod1, nullptr);

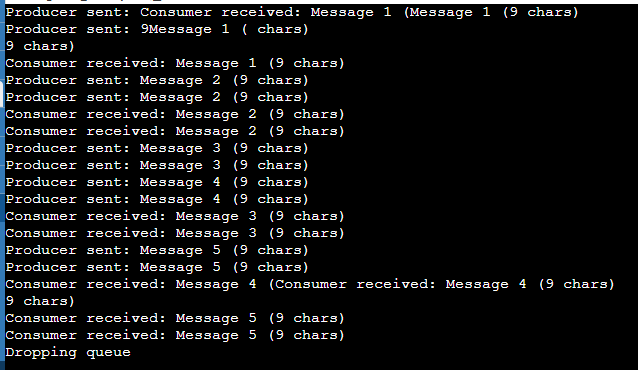
pthread\_join(prod2, nullptr);

pthread\_join(cons1, nullptr);

pthread\_join(cons2, nullptr);

return 0;

}



**Задание 26**:Производитель-потребитель 2

**Условие**: Реализуйте задачу 25 с использованием условных переменных и минимально необходимого числа мутексов.

#include <iostream>

#include <deque>

#include <string>

#include <pthread.h>

#include <cstring>

#include <unistd.h>

class MessageQueue {

private: std::deque < std::string > queue;

pthread\_mutex\_t mutex;

pthread\_cond\_t cond\_slots; // Для ожидания свободных слотов

pthread\_cond\_t cond\_items; // Для ожидания сообщений

bool dropped;

public: MessageQueue(): dropped(false) {

pthread\_mutex\_init( & mutex, nullptr);

pthread\_cond\_init( & cond\_slots, nullptr);

pthread\_cond\_init( & cond\_items, nullptr);

}

~MessageQueue() {

mymsgdestroy();

}

int mymsgput(const char \* msg) {

pthread\_mutex\_lock( & mutex);

if (dropped) {

pthread\_mutex\_unlock( & mutex);

return 0;

}

// Обрезаем строку до 80 символов

std::string str(msg);

if (str.size() > 80) str = str.substr(0, 80);

int len = str.size();

// Ждём, пока в очереди < 10 элементов

while (queue.size() >= 10 && !dropped) {

pthread\_cond\_wait( & cond\_slots, & mutex);

}

if (dropped) {

pthread\_mutex\_unlock( & mutex);

return 0;

}

queue.push\_back(str);

pthread\_cond\_signal( & cond\_items); // Сигнализируем о новом сообщении

pthread\_mutex\_unlock( & mutex);

return len;

}

int mymsgget(char \* buf, size\_t bufsize) {

pthread\_mutex\_lock( & mutex);

if (dropped) {

pthread\_mutex\_unlock( & mutex);

return 0;

}

// Ждём, пока очередь не пуста

while (queue.empty() && !dropped) {

pthread\_cond\_wait( & cond\_items, & mutex);

}

if (dropped) {

pthread\_mutex\_unlock( & mutex);

return 0;

}

std::string msg = queue.front();

queue.pop\_front();

// Обрезаем до размера буфера

size\_t copy\_len = std::min(msg.size(), bufsize - 1);

strncpy(buf, msg.c\_str(), copy\_len);

buf[copy\_len] = '\0';

pthread\_cond\_signal( & cond\_slots); // Сигнализируем о свободном слоте

pthread\_mutex\_unlock( & mutex);

return copy\_len;

}

void mymsgdrop() {

pthread\_mutex\_lock( & mutex);

dropped = true;

queue.clear();

pthread\_cond\_broadcast( & cond\_slots); // Разблокируем put

pthread\_cond\_broadcast( & cond\_items); // Разблокируем get

pthread\_mutex\_unlock( & mutex);

}

void mymsgdestroy() {

mymsgdrop();

pthread\_mutex\_destroy( & mutex);

pthread\_cond\_destroy( & cond\_slots);

pthread\_cond\_destroy( & cond\_items);

}

};

// Потоки-производители и потребители (идентичны заданию 25)

void \* producer(void \* arg) {

MessageQueue \* q = (MessageQueue \* ) arg;

for (int i = 1; i <= 5; ++i) {

std::string msg = "Message " + std::to\_string(i);

int len = q -> mymsgput(msg.c\_str());

if (len == 0) break;

std::cout << "Producer sent: " << msg << " (" << len << " chars)\n";

sleep(1);

}

return nullptr;

}

void \* consumer(void \* arg) {

MessageQueue \* q = (MessageQueue \* ) arg;

char buf[100];

while (true) {

int len = q -> mymsgget(buf, sizeof(buf));

if (len == 0) break;

std::cout << "Consumer received: " << buf << " (" << len << " chars)\n";

sleep(2);

}

return nullptr;

}

int main() {

MessageQueue q;

// Создание потоков

pthread\_t prod1, prod2, cons1, cons2;

pthread\_create( & prod1, nullptr, producer, & q);

pthread\_create( & prod2, nullptr, producer, & q);

pthread\_create( & cons1, nullptr, consumer, & q);

pthread\_create( & cons2, nullptr, consumer, & q);

// Ждём 10 секунд, затем сбрасываем очередь

sleep(10);

std::cout << "Dropping queue\n";

q.mymsgdrop();

// Ждём завершения потоков

pthread\_join(prod1, nullptr);

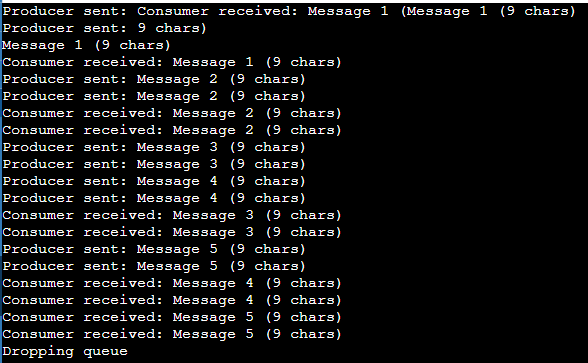
pthread\_join(prod2, nullptr);

pthread\_join(cons1, nullptr);

pthread\_join(cons2, nullptr);

return 0;

}



**Часть 2**

**Задание 27**:Многопоточный сервер

**Условие:** Реализуйте сервер, который принимает ТСР соединения и транслирует их. Сервер должен получать из командной строки следующие параметры:

* Номер порта Р, на котором следует слушать.
* Имя или ІР-адрес узла N, на который следует транслировать соединения.
* Номер порта Р', на который следует транслировать соединения.

Сервер принимает все входящие запросы на установление соединения на порт Р. Для каждого такого соединения он открывает соединение с портом Р' на сервере N. Затем он транслирует все данные, получаемые от клиента, серверу N, а все данные, получаемые от сервера N - клиенту. Если сервер N или клиент разрывают соединение, наш сервер также должен разорвать соединение. Если сервер N отказывает в установлении соединения, следует разорвать клиентское соединение.

Сервер должен обеспечивать трансляцию 510 соединений при лимите количества открытых файлов на процесс 1024. Сервер не должен быть многопоточным и никогда не должен блокироваться при операциях чтения и записи. Не следует использовать неблокирующиеся сокеты. Следует использовать select или poll.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <poll.h>

#include <vector>

#include <errno.h>

const int MAX\_CONNECTIONS = 510; // Максимум соединений

const int BUFFER\_SIZE = 4096; // Размер буфера

struct Connection {

int client\_fd; // Клиентский сокет

int server\_fd; // Серверный сокет

bool active; // Флаг активности

Connection(): client\_fd(-1), server\_fd(-1), active(false) {}

};

// Установка соединения с сервером N:P'

int connect\_to\_server(const char \* node,

const char \* port) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(node, port, & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int server\_fd = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (server\_fd < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(server\_fd, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(server\_fd);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return server\_fd;

}

// Закрытие соединения

void close\_connection(Connection & conn) {

if (conn.client\_fd >= 0) {

close(conn.client\_fd);

conn.client\_fd = -1;

}

if (conn.server\_fd >= 0) {

close(conn.server\_fd);

conn.server\_fd = -1;

}

conn.active = false;

}

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

if (argc != 4) {

std::cerr << "Usage: " << argv[0] << " <listen\_port> <node> <node\_port>" << std::endl;

return 1;

}

int listen\_port = std::atoi(argv[1]);

const char \* node = argv[2];

const char \* node\_port = argv[3];

// Создание слушающего сокета

int listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (listen\_fd < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

return 1;

}

// Настройка адреса

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = INADDR\_ANY;

addr.sin\_port = htons(listen\_port);

// Разрешаем повторное использование порта

int opt = 1;

setsockopt(listen\_fd, SOL\_SOCKET, SO\_REUSEADDR, & opt, sizeof(opt));

if (bind(listen\_fd, (struct sockaddr \* ) & addr, sizeof(addr)) < 0) {

std::cerr << "bind failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

if (listen(listen\_fd, 10) < 0) {

std::cerr << "listen failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

std::cout << "Server listening on port " << listen\_port << ", forwarding to " << node << ":" << node\_port << std::endl;

// Массив соединений

std::vector < Connection > connections(MAX\_CONNECTIONS);

std::vector < struct pollfd > fds;

fds.push\_back({

listen\_fd,

POLLIN,

0

}); // Слушающий сокет

char buffer[BUFFER\_SIZE];

while (true) {

// Ожидание событий с poll

if (poll(fds.data(), fds.size(), -1) < 0) {

std::cerr << "poll failed: " << strerror(errno) << std::endl;

break;

}

// Обработка событий

for (size\_t i = 0; i < fds.size(); ++i) {

if (fds[i].revents == 0) continue;

if (fds[i].fd == listen\_fd) {

// Новое клиентское соединение

struct sockaddr\_in client\_addr;

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

int client\_fd = accept(listen\_fd, (struct sockaddr \* ) & client\_addr, & client\_len);

if (client\_fd < 0) {

std::cerr << "accept failed: " << strerror(errno) << std::endl;

continue;

}

// Подключение к серверу N

int server\_fd = connect\_to\_server(node, node\_port);

if (server\_fd < 0) {

close(client\_fd);

continue;

}

// Найти свободное место для соединения

bool found = false;

for (int j = 0; j < MAX\_CONNECTIONS; ++j) {

if (!connections[j].active) {

connections[j].client\_fd = client\_fd;

connections[j].server\_fd = server\_fd;

connections[j].active = true;

fds.push\_back({

client\_fd,

POLLIN,

0

});

fds.push\_back({

server\_fd,

POLLIN,

0

});

found = true;

break;

}

}

if (!found) {

std::cerr << "Max connections reached" << std::endl;

close(client\_fd);

close(server\_fd);

}

} else {

// Данные от клиента или сервера

int fd = fds[i].fd;

bool is\_client = false;

int conn\_idx = -1;

for (int j = 0; j < MAX\_CONNECTIONS; ++j) {

if (connections[j].active && (connections[j].client\_fd == fd || connections[j].server\_fd == fd)) {

is\_client = (fd == connections[j].client\_fd);

conn\_idx = j;

break;

}

}

if (conn\_idx == -1) continue;

int src\_fd = is\_client ? connections[conn\_idx].client\_fd : connections[conn\_idx].server\_fd;

int dst\_fd = is\_client ? connections[conn\_idx].server\_fd : connections[conn\_idx].client\_fd;

ssize\_t n = read(src\_fd, buffer, BUFFER\_SIZE);

if (n <= 0) {

// Соединение закрыто или ошибка

close\_connection(connections[conn\_idx]);

// Удаляем сокеты из fds

fds.erase(fds.begin() + i);

for (size\_t j = 0; j < fds.size(); ++j) {

if (fds[j].fd == dst\_fd) {

fds.erase(fds.begin() + j);

break;

}

}

continue;

}

// Пересылка данных

ssize\_t written = write(dst\_fd, buffer, n);

if (written != n) {

std::cerr << "write failed: " << strerror(errno) << std::endl;

close\_connection(connections[conn\_idx]);

fds.erase(fds.begin() + i);

for (size\_t j = 0; j < fds.size(); ++j) {

if (fds[j].fd == dst\_fd) {

fds.erase(fds.begin() + j);

break;

}

}

}

}

}

}

// Очистка

close(listen\_fd);

for (auto & conn: connections) {

close\_connection(conn);

}

return 0;

}

**Задание 28**:Псевдомногопоточный НТТР-клиент

**Условие:** Реализуйте простой НТТР-клиент. Он принимает один параметр командной строки - URL. Клиент делает запрос по указанному URL и выдает тело ответа на терминал как текст (т.е. если в ответе НТМL, то распечатывает его исходный текст без форматирования). Вывод производится по мере того, как данные поступают из НТТР-соединения. Когда будет выведено более экрана (более 25 строк) данных, клиент должен продолжить прием данных, но должен остановить вывод и выдать приглашение Press space to scroll down.

При нажатии пользователем клиент должен вывести следующий экран данных. Для одновременного считывания данных с терминала и из сетевого соединения используйте системный вызов select.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <sys/select.h>

#include <vector>

#include <string>

const int BUFFER\_SIZE = 4096;

const int LINES\_PER\_SCREEN = 25;

// Парсинг URL (простая версия: http://host[:port]/path)

bool parse\_url(const std::string & url, std::string & host, std::string & port, std::string & path) {

if (url.substr(0, 7) != "http://") return false;

std::string rest = url.substr(7);

size\_t slash\_pos = rest.find('/');

size\_t colon\_pos = rest.find(':');

if (slash\_pos == std::string::npos) {

path = "/";

if (colon\_pos == std::string::npos) {

host = rest;

port = "80";

} else {

host = rest.substr(0, colon\_pos);

port = rest.substr(colon\_pos + 1);

}

} else {

path = rest.substr(slash\_pos);

if (colon\_pos == std::string::npos || colon\_pos > slash\_pos) {

host = rest.substr(0, slash\_pos);

port = "80";

} else {

host = rest.substr(0, colon\_pos);

port = rest.substr(colon\_pos + 1, slash\_pos - colon\_pos - 1);

}

}

return true;

}

// Подключение к серверу

int connect\_to\_server(const std::string & host,

const std::string & port) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host.c\_str(), port.c\_str(), & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int sock = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (sock < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(sock, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(sock);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return sock;

}

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

if (argc != 2) {

std::cerr << "Usage: " << argv[0] << " <URL>" << std::endl;

return 1;

}

std::string url = argv[1];

std::string host, port, path;

if (!parse\_url(url, host, port, path)) {

std::cerr << "Invalid URL" << std::endl;

return 1;

}

// Подключение к серверу

int sock = connect\_to\_server(host, port);

if (sock < 0) return 1;

// Отправка GET-запроса

std::string request = "GET " + path + " HTTP/1.1\r\nHost: " + host + "\r\nConnection: close\r\n\r\n";

if (write(sock, request.c\_str(), request.size()) < 0) {

std::cerr << "write failed: " << strerror(errno) << std::endl;

close(sock);

return 1;

}

char buffer[BUFFER\_SIZE];

bool headers\_done = false;

std::string body;

int line\_count = 0;

bool waiting\_for\_space = false;

fd\_set read\_fds;

struct timeval tv;

while (true) {

FD\_ZERO( & read\_fds);

FD\_SET(STDIN\_FILENO, & read\_fds);

FD\_SET(sock, & read\_fds);

int max\_fd = std::max(STDIN\_FILENO, sock) + 1;

tv.tv\_sec = 5;

tv.tv\_usec = 0;

int ret = select(max\_fd, & read\_fds, nullptr, nullptr, & tv);

if (ret < 0) {

std::cerr << "select failed: " << strerror(errno) << std::endl;

break;

}

if (FD\_ISSET(sock, & read\_fds)) {

ssize\_t n = read(sock, buffer, BUFFER\_SIZE - 1);

if (n <= 0) {

if (n < 0) std::cerr << "read failed: " << strerror(errno) << std::endl;

break;

}

buffer[n] = '\0';

if (!headers\_done) {

body += buffer;

size\_t header\_end = body.find("\r\n\r\n");

if (header\_end != std::string::npos) {

body = body.substr(header\_end + 4);

headers\_done = true;

}

} else {

body += buffer;

}

if (headers\_done && !waiting\_for\_space) {

size\_t pos = 0;

while (pos < body.size()) {

size\_t next = body.find('\n', pos);

if (next == std::string::npos) break;

std::string line = body.substr(pos, next - pos + 1);

std::cout << line;

line\_count++;

pos = next + 1;

if (line\_count >= LINES\_PER\_SCREEN) {

std::cout << "Press space to scroll down" << std::endl;

waiting\_for\_space = true;

break;

}

}

if (pos < body.size()) {

body = body.substr(pos);

} else {

body.clear();

}

}

}

if (FD\_ISSET(STDIN\_FILENO, & read\_fds) && waiting\_for\_space) {

char input;

if (read(STDIN\_FILENO, & input, 1) > 0 && input == ' ') {

waiting\_for\_space = false;

line\_count = 0;

size\_t pos = 0;

while (pos < body.size()) {

size\_t next = body.find('\n', pos);

if (next == std::string::npos) break;

std::string line = body.substr(pos, next - pos + 1);

std::cout << line;

line\_count++;

pos = next + 1;

if (line\_count >= LINES\_PER\_SCREEN) {

std::cout << "Press space to scroll down" << std::endl;

waiting\_for\_space = true;

break;

}

}

if (pos < body.size()) {

body = body.substr(pos);

} else {

body.clear();

}

}

}

}

close(sock);

return 0;

}

**Задание 29**:Псевдомногопоточный НТТР-клиент 2

**Условие:** Реализуйте задачу упр. 28, используя системные вызовы aio\_read/aio\_write.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <aio.h>

#include <fcntl.h>

#include <string>

#include <vector>

const int BUFFER\_SIZE = 4096;

const int LINES\_PER\_SCREEN = 25;

// Парсинг URL (идентичен заданию 28)

bool parse\_url(const std::string & url, std::string & host, std::string & port, std::string & path) {

if (url.substr(0, 7) != "http://") return false;

std::string rest = url.substr(7);

size\_t slash\_pos = rest.find('/');

size\_t colon\_pos = rest.find(':');

if (slash\_pos == std::string::npos) {

path = "/";

if (colon\_pos == std::string::npos) {

host = rest;

port = "80";

} else {

host = rest.substr(0, colon\_pos);

port = rest.substr(colon\_pos + 1);

}

} else {

path = rest.substr(slash\_pos);

if (colon\_pos == std::string::npos || colon\_pos > slash\_pos) {

host = rest.substr(0, slash\_pos);

port = "80";

} else {

host = rest.substr(0, colon\_pos);

port = rest.substr(colon\_pos + 1, slash\_pos - colon\_pos - 1);

}

}

return true;

}

// Подключение к серверу (идентичен заданию 28)

int connect\_to\_server(const std::string & host,

const std::string & port) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host.c\_str(), port.c\_str(), & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int sock = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (sock < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(sock, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(sock);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return sock;

}

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

if (argc != 2) {

std::cerr << "Usage: " << argv[0] << " <URL>" << std::endl;

return 1;

}

std::string url = argv[1];

std::string host, port, path;

if (!parse\_url(url, host, port, path)) {

std::cerr << "Invalid URL" << std::endl;

return 1;

}

// Подключение к серверу

int sock = connect\_to\_server(host, port);

if (sock < 0) return 1;

// Установка неблокирующего режима для AIO

fcntl(sock, F\_SETFL, fcntl(sock, F\_GETFL) | O\_ASYNC);

fcntl(STDIN\_FILENO, F\_SETFL, fcntl(STDIN\_FILENO, F\_GETFL) | O\_ASYNC);

// Отправка GET-запроса

std::string request = "GET " + path + " HTTP/1.1\r\nHost: " + host + "\r\nConnection: close\r\n\r\n";

if (write(sock, request.c\_str(), request.size()) < 0) {

std::cerr << "write failed: " << strerror(errno) << std::endl;

close(sock);

return 1;

}

char sock\_buffer[BUFFER\_SIZE];

char stdin\_buffer[1];

bool headers\_done = false;

std::string body;

int line\_count = 0;

bool waiting\_for\_space = false;

// Настройка AIO для сокета

struct aiocb sock\_cb;

memset( & sock\_cb, 0, sizeof(sock\_cb));

sock\_cb.aio\_fildes = sock;

sock\_cb.aio\_buf = sock\_buffer;

sock\_cb.aio\_nbytes = BUFFER\_SIZE - 1;

sock\_cb.aio\_offset = 0;

// Настройка AIO для stdin

struct aiocb stdin\_cb;

memset( & stdin\_cb, 0, sizeof(stdin\_cb));

stdin\_cb.aio\_fildes = STDIN\_FILENO;

stdin\_cb.aio\_buf = stdin\_buffer;

stdin\_cb.aio\_nbytes = 1;

stdin\_cb.aio\_offset = 0;

// Инициализация асинхронного чтения

if (!waiting\_for\_space) {

aio\_read( & sock\_cb);

}

if (waiting\_for\_space) {

aio\_read( & stdin\_cb);

}

while (true) {

struct aiocb \* cblist[] = {

& sock\_cb,

& stdin\_cb

};

if (aio\_suspend(cblist, 2, nullptr) < 0) {

std::cerr << "aio\_suspend failed: " << strerror(errno) << std::endl;

break;

}

// Проверка завершения чтения из сокета

if (aio\_error( & sock\_cb) == 0) {

ssize\_t n = aio\_return( & sock\_cb);

if (n <= 0) {

if (n < 0) std::cerr << "aio\_read failed: " << strerror(errno) << std::endl;

break;

}

sock\_buffer[n] = '\0';

if (!headers\_done) {

body += sock\_buffer;

size\_t header\_end = body.find("\r\n\r\n");

if (header\_end != std::string::npos) {

body = body.substr(header\_end + 4);

headers\_done = true;

}

} else {

body += sock\_buffer;

}

if (headers\_done && !waiting\_for\_space) {

size\_t pos = 0;

while (pos < body.size()) {

size\_t next = body.find('\n', pos);

if (next == std::string::npos) break;

std::string line = body.substr(pos, next - pos + 1);

std::cout << line;

line\_count++;

pos = next + 1;

if (line\_count >= LINES\_PER\_SCREEN) {

std::cout << "Press space to scroll down" << std::endl;

waiting\_for\_space = true;

aio\_read( & stdin\_cb);

break;

}

}

if (pos < body.size()) {

body = body.substr(pos);

} else {

body.clear();

}

}

if (!waiting\_for\_space) {

aio\_read( & sock\_cb);

}

}

// Проверка завершения чтения из stdin

if (aio\_error( & stdin\_cb) == 0 && waiting\_for\_space) {

ssize\_t n = aio\_return( & stdin\_cb);

if (n > 0 && stdin\_buffer[0] == ' ') {

waiting\_for\_space = false;

line\_count = 0;

size\_t pos = 0;

while (pos < body.size()) {

size\_t next = body.find('\n', pos);

if (next == std::string::npos) break;

std::string line = body.substr(pos, next - pos + 1);

std::cout << line;

line\_count++;

pos = next + 1;

if (line\_count >= LINES\_PER\_SCREEN) {

std::cout << "Press space to scroll down" << std::endl;

waiting\_for\_space = true;

aio\_read( & stdin\_cb);

break;

}

}

if (pos < body.size()) {

body = body.substr(pos);

} else {

body.clear();

}

if (!waiting\_for\_space) {

aio\_read( & sock\_cb);

}

}

}

}

close(sock);

return 0;

}

**Задание 30**:Многопоточный НТТР-клиент

**Условие:** Реализуйте задачу упр. 28, используя два потока, одну для считывания данных из сетевого соединения, другую для взаимодействия с пользователем.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <pthread.h>

#include <string>

#include <vector>

#include <queue>

const int BUFFER\_SIZE = 4096;

const int LINES\_PER\_SCREEN = 25;

// Парсинг URL (идентичен заданию 28)

bool parse\_url(const std::string & url, std::string & host, std::string & port, std::string & path) {

if (url.substr(0, 7) != "http://") return false;

std::string rest = url.substr(7);

size\_t slash\_pos = rest.find('/');

size\_t colon\_pos = rest.find(':');

if (slash\_pos == std::string::npos) {

path = "/";

if (colon\_pos == std::string::npos) {

host = rest;

port = "80";

} else {

host = rest.substr(0, colon\_pos);

port = rest.substr(colon\_pos + 1);

}

} else {

path = rest.substr(slash\_pos);

if (colon\_pos == std::string::npos || colon\_pos > slash\_pos) {

host = rest.substr(0, slash\_pos);

port = "80";

} else {

host = rest.substr(0, colon\_pos);

port = rest.substr(colon\_pos + 1, slash\_pos - colon\_pos - 1);

}

}

return true;

}

// Подключение к серверу (идентичен заданию 28)

int connect\_to\_server(const std::string & host,

const std::string & port) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host.c\_str(), port.c\_str(), & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int sock = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (sock < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(sock, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(sock);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return sock;

}

struct ThreadData {

int sock;

std::queue < std::string > \* lines;

pthread\_mutex\_t \* mutex;

pthread\_cond\_t \* cond;

bool \* done;

bool \* headers\_done;

std::string \* body;

};

// Сетевой поток: чтение данных из сокета

void \* network\_thread(void \* arg) {

ThreadData \* data = (ThreadData \* ) arg;

char buffer[BUFFER\_SIZE];

while (true) {

ssize\_t n = read(data -> sock, buffer, BUFFER\_SIZE - 1);

if (n <= 0) {

if (n < 0) std::cerr << "read failed: " << strerror(errno) << std::endl;

pthread\_mutex\_lock(data -> mutex);

\*(data -> done) = true;

pthread\_cond\_signal(data -> cond);

pthread\_mutex\_unlock(data -> mutex);

break;

}

buffer[n] = '\0';

pthread\_mutex\_lock(data -> mutex);

if (! \* (data -> headers\_done)) {

\*(data -> body) += buffer;

size\_t header\_end = data -> body -> find("\r\n\r\n");

if (header\_end != std::string::npos) {

\*(data -> body) = data -> body -> substr(header\_end + 4);

\*(data -> headers\_done) = true;

}

} else {

\*(data -> body) += buffer;

}

if ( \* (data -> headers\_done)) {

size\_t pos = 0;

while (pos < data -> body -> size()) {

size\_t next = data -> body -> find('\n', pos);

if (next == std::string::npos) break;

data -> lines -> push(data -> body -> substr(pos, next - pos + 1));

pos = next + 1;

}

if (pos < data -> body -> size()) {

\*(data -> body) = data -> body -> substr(pos);

} else {

data -> body -> clear();

}

pthread\_cond\_signal(data -> cond);

}

pthread\_mutex\_unlock(data -> mutex);

}

return nullptr;

}

// Пользовательский поток: вывод данных и взаимодействие

void \* user\_thread(void \* arg) {

ThreadData \* data = (ThreadData \* ) arg;

int line\_count = 0;

bool waiting\_for\_space = false;

while (true) {

pthread\_mutex\_lock(data -> mutex);

while (data -> lines -> empty() && ! \* (data -> done)) {

pthread\_cond\_wait(data -> cond, data -> mutex);

}

if (data -> lines -> empty() && \* (data -> done)) {

pthread\_mutex\_unlock(data -> mutex);

break;

}

if (!waiting\_for\_space) {

while (!data -> lines -> empty() && line\_count < LINES\_PER\_SCREEN) {

std::cout << data -> lines -> front();

data -> lines -> pop();

line\_count++;

}

if (line\_count >= LINES\_PER\_SCREEN) {

std::cout << "Press space to scroll down" << std::endl;

waiting\_for\_space = true;

}

}

pthread\_mutex\_unlock(data -> mutex);

if (waiting\_for\_space) {

char input;

if (read(STDIN\_FILENO, & input, 1) > 0 && input == ' ') {

pthread\_mutex\_lock(data -> mutex);

waiting\_for\_space = false;

line\_count = 0;

pthread\_mutex\_unlock(data -> mutex);

}

}

}

return nullptr;

}

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

if (argc != 2) {

std::cerr << "Usage: " << argv[0] << " <URL>" << std::endl;

return 1;

}

std::string url = argv[1];

std::string host, port, path;

if (!parse\_url(url, host, port, path)) {

std::cerr << "Invalid URL" << std::endl;

return 1;

}

// Подключение к серверу

int sock = connect\_to\_server(host, port);

if (sock < 0) return 1;

// Отправка GET-запроса

std::string request = "GET " + path + " HTTP/1.1\r\nHost: " + host + "\r\nConnection: close\r\n\r\n";

if (write(sock, request.c\_str(), request.size()) < 0) {

std::cerr << "write failed: " << strerror(errno) << std::endl;

close(sock);

return 1;

}

// Подготовка данных для потоков

std::queue < std::string > lines;

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

bool done = false;

bool headers\_done = false;

std::string body;

pthread\_mutex\_init( & mutex, nullptr);

pthread\_cond\_init( & cond, nullptr);

ThreadData data = {

sock,

& lines,

& mutex,

& cond,

& done,

& headers\_done,

& body

};

// Создание потоков

pthread\_t network, user;

pthread\_create( & network, nullptr, network\_thread, & data);

pthread\_create( & user, nullptr, user\_thread, & data);

// Ожидание завершения

pthread\_join(network, nullptr);

pthread\_join(user, nullptr);

// Очистка

pthread\_mutex\_destroy( & mutex);

pthread\_cond\_destroy( & cond);

close(sock);

return 0;

}

**Задание 31**:Псевдомногопоточный кэширующий прокси

**Условие:** Реализуйте простой кэширующий HTTP-рrоxу с кэшем в оперативной памяти.

Прокси должен быть реализован как один процесс и один поток, использующий для одновременной работы с несколькими сетевыми соединениями системный вызов select или poll. Прокси должен обеспечивать одновременную работу нескольких клиентов (один клиент не должен ждать завершения запроса или этапа обработки запроса другого клиента).

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <poll.h>

#include <map>

#include <string>

#include <vector>

const int BUFFER\_SIZE = 4096;

const int MAX\_CLIENTS = 100;

const int LISTEN\_PORT = 8080;

struct CacheEntry {

std::string response;

};

struct Connection {

int client\_fd;

int server\_fd;

std::string request;

std::string response;

bool headers\_sent;

bool is\_client;

std::string host;

std::string path;

Connection(): client\_fd(-1), server\_fd(-1), headers\_sent(false), is\_client(true) {}

};

// Парсинг HTTP-запроса

bool parse\_request(const std::string & req, std::string & host, std::string & path) {

size\_t get\_pos = req.find("GET ");

if (get\_pos == std::string::npos) return false;

size\_t http\_pos = req.find(" HTTP/1.");

if (http\_pos == std::string::npos) return false;

std::string url = req.substr(get\_pos + 4, http\_pos - get\_pos - 4);

if (url.substr(0, 7) == "http://") {

url = url.substr(7);

}

size\_t slash\_pos = url.find('/');

if (slash\_pos == std::string::npos) {

host = url;

path = "/";

} else {

host = url.substr(0, slash\_pos);

path = url.substr(slash\_pos);

}

return true;

}

// Подключение к серверу

int connect\_to\_server(const std::string & host) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host.c\_str(), "80", & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int sock = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (sock < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(sock, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(sock);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return sock;

}

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

// Создание слушающего сокета

int listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (listen\_fd < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

return 1;

}

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = INADDR\_ANY;

addr.sin\_port = htons(LISTEN\_PORT);

int opt = 1;

setsockopt(listen\_fd, SOL\_SOCKET, SO\_REUSEADDR, & opt, sizeof(opt));

if (bind(listen\_fd, (struct sockaddr \* ) & addr, sizeof(addr)) < 0) {

std::cerr << "bind failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

if (listen(listen\_fd, 10) < 0) {

std::cerr << "listen failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

std::cout << "Proxy listening on port " << LISTEN\_PORT << std::endl;

std::vector < Connection > connections(MAX\_CLIENTS);

std::vector < struct pollfd > fds;

fds.push\_back({

listen\_fd,

POLLIN,

0

});

std::map < std::string, CacheEntry > cache;

char buffer[BUFFER\_SIZE];

while (true) {

if (poll(fds.data(), fds.size(), -1) < 0) {

std::cerr << "poll failed: " << strerror(errno) << std::endl;

break;

}

for (size\_t i = 0; i < fds.size(); ++i) {

if (fds[i].revents == 0) continue;

if (fds[i].fd == listen\_fd) {

struct sockaddr\_in client\_addr;

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

int client\_fd = accept(listen\_fd, (struct sockaddr \* ) & client\_addr, & client\_len);

if (client\_fd < 0) {

std::cerr << "accept failed: " << strerror(errno) << std::endl;

continue;

}

bool found = false;

for (size\_t j = 0; j < connections.size(); ++j) {

if (connections[j].client\_fd == -1) {

connections[j].client\_fd = client\_fd;

connections[j].is\_client = true;

fds.push\_back({

client\_fd,

POLLIN,

0

});

found = true;

break;

}

}

if (!found) {

std::cerr << "Max clients reached" << std::endl;

close(client\_fd);

}

} else {

int fd = fds[i].fd;

size\_t conn\_idx = 0;

bool is\_client = false;

for (size\_t j = 0; j < connections.size(); ++j) {

if (connections[j].client\_fd == fd) {

conn\_idx = j;

is\_client = true;

break;

} else if (connections[j].server\_fd == fd) {

conn\_idx = j;

break;

}

}

Connection & conn = connections[conn\_idx];

ssize\_t n = read(fd, buffer, BUFFER\_SIZE - 1);

if (n <= 0) {

if (is\_client) {

close(conn.client\_fd);

if (conn.server\_fd >= 0) close(conn.server\_fd);

conn.client\_fd = -1;

conn.server\_fd = -1;

} else {

close(conn.server\_fd);

conn.server\_fd = -1;

}

fds.erase(fds.begin() + i);

--i;

continue;

}

buffer[n] = '\0';

if (is\_client) {

conn.request += buffer;

size\_t header\_end = conn.request.find("\r\n\r\n");

if (header\_end != std::string::npos) {

if (!parse\_request(conn.request, conn.host, conn.path)) {

std::cerr << "Invalid request" << std::endl;

close(conn.client\_fd);

conn.client\_fd = -1;

fds.erase(fds.begin() + i);

--i;

continue;

}

std::string cache\_key = conn.host + conn.path;

if (cache.find(cache\_key) != cache.end()) {

conn.response = cache[cache\_key].response;

conn.headers\_sent = true;

write(conn.client\_fd, conn.response.c\_str(), conn.response.size());

close(conn.client\_fd);

conn.client\_fd = -1;

fds.erase(fds.begin() + i);

--i;

continue;

}

conn.server\_fd = connect\_to\_server(conn.host);

if (conn.server\_fd < 0) {

close(conn.client\_fd);

conn.client\_fd = -1;

fds.erase(fds.begin() + i);

--i;

continue;

}

fds.push\_back({

conn.server\_fd,

POLLIN,

0

});

write(conn.server\_fd, conn.request.c\_str(), conn.request.size());

}

} else {

conn.response += buffer;

if (!conn.headers\_sent) {

size\_t header\_end = conn.response.find("\r\n\r\n");

if (header\_end != std::string::npos) {

conn.headers\_sent = true;

write(conn.client\_fd, conn.response.c\_str(), conn.response.size());

std::string cache\_key = conn.host + conn.path;

cache[cache\_key].response = conn.response;

}

} else {

write(conn.client\_fd, buffer, n);

std::string cache\_key = conn.host + conn.path;

cache[cache\_key].response += buffer;

}

}

}

}

}

close(listen\_fd);

for (auto & conn: connections) {

if (conn.client\_fd >= 0) close(conn.client\_fd);

if (conn.server\_fd >= 0) close(conn.server\_fd);

}

return 0;

}

**Задание 32**:Многопоточный кэширующий прокси

**Условие:** Реализовать задачу 31, создавая для каждого входящего НТТР-соединения свой поток. При невозможности создать поток допускается блокировать входящие соединения или возвращать ошибку.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <pthread.h>

#include <map>

#include <string>

const int BUFFER\_SIZE = 4096;

const int LISTEN\_PORT = 8080;

struct CacheEntry {

std::string response;

};

std::map < std::string, CacheEntry > cache;

pthread\_mutex\_t cache\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

// Парсинг HTTP-запроса (идентичен заданию 31)

bool parse\_request(const std::string & req, std::string & host, std::string & path) {

size\_t get\_pos = req.find("GET ");

if (get\_pos == std::string::npos) return false;

size\_t http\_pos = req.find(" HTTP/1.");

if (http\_pos == std::string::npos) return false;

std::string url = req.substr(get\_pos + 4, http\_pos - get\_pos - 4);

if (url.substr(0, 7) == "http://") {

url = url.substr(7);

}

size\_t slash\_pos = url.find('/');

if (slash\_pos == std::string::npos) {

host = url;

path = "/";

} else {

host = url.substr(0, slash\_pos);

path = url.substr(slash\_pos);

}

return true;

}

// Подключение к серверу (идентичен заданию 31)

int connect\_to\_server(const std::string & host) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host.c\_str(), "80", & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int sock = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (sock < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(sock, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(sock);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return sock;

}

// Обработка клиента в отдельном потоке

void \* handle\_client(void \* arg) {

int client\_fd = \* (int \* ) arg;

delete(int \* ) arg;

char buffer[BUFFER\_SIZE];

std::string request;

while (true) {

ssize\_t n = read(client\_fd, buffer, BUFFER\_SIZE - 1);

if (n <= 0) {

close(client\_fd);

return nullptr;

}

buffer[n] = '\0';

request += buffer;

if (request.find("\r\n\r\n") != std::string::npos) break;

}

std::string host, path;

if (!parse\_request(request, host, path)) {

std::cerr << "Invalid request" << std::endl;

close(client\_fd);

return nullptr;

}

std::string cache\_key = host + path;

std::string response;

pthread\_mutex\_lock( & cache\_mutex);

if (cache.find(cache\_key) != cache.end()) {

response = cache[cache\_key].response;

pthread\_mutex\_unlock( & cache\_mutex);

write(client\_fd, response.c\_str(), response.size());

close(client\_fd);

return nullptr;

}

pthread\_mutex\_unlock( & cache\_mutex);

int server\_fd = connect\_to\_server(host);

if (server\_fd < 0) {

close(client\_fd);

return nullptr;

}

write(server\_fd, request.c\_str(), request.size());

while (true) {

ssize\_t n = read(server\_fd, buffer, BUFFER\_SIZE);

if (n <= 0) break;

response += std::string(buffer, n);

write(client\_fd, buffer, n);

}

pthread\_mutex\_lock( & cache\_mutex);

cache[cache\_key].response = response;

pthread\_mutex\_unlock( & cache\_mutex);

close(server\_fd);

close(client\_fd);

return nullptr;

}

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

int listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (listen\_fd < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

return 1;

}

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = INADDR\_ANY;

addr.sin\_port = htons(LISTEN\_PORT);

int opt = 1;

setsockopt(listen\_fd, SOL\_SOCKET, SO\_REUSEADDR, & opt, sizeof(opt));

if (bind(listen\_fd, (struct sockaddr \* ) & addr, sizeof(addr)) < 0) {

std::cerr << "bind failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

if (listen(listen\_fd, 10) < 0) {

std::cerr << "listen failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

std::cout << "Proxy listening on port " << LISTEN\_PORT << std::endl;

while (true) {

struct sockaddr\_in client\_addr;

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

int client\_fd = accept(listen\_fd, (struct sockaddr \* ) & client\_addr, & client\_len);

if (client\_fd < 0) {

std::cerr << "accept failed: " << strerror(errno) << std::endl;

continue;

}

pthread\_t thread;

int \* client\_fd\_ptr = new int(client\_fd);

if (pthread\_create( & thread, nullptr, handle\_client, client\_fd\_ptr) != 0) {

std::cerr << "pthread\_create failed: " << strerror(errno) << std::endl;

close(client\_fd);

delete client\_fd\_ptr;

continue;

}

pthread\_detach(thread);

}

close(listen\_fd);

pthread\_mutex\_destroy( & cache\_mutex);

return 0;

}

**Задание 33**:Многопоточный кэширующий прокси с рабочими потоками

**Условие:** Реализовать задачу 31, используя рабочие потоки (worked threads). При запуске прокси должен принимать параметр, целое число, указывающее размер пула потоков. Прокси должен запустить указанное число поток. Необходимо обеспечить одновременную обработку количества запросов, превосходящего количество потоков в пуле; блокировка входящих соединений недопустима. Разумеется, при этом каждый из потоков в разные моменты времени будет вынужден обрабатывать разные соединения. Для управления соединениями используйте select или роll.

#include <iostream>

#include <cstring>

#include <cstdlib>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <poll.h>

#include <pthread.h>

#include <queue>

#include <map>

#include <string>

#include <vector>

const int BUFFER\_SIZE = 4096;

const int LISTEN\_PORT = 8080;

const int MAX\_CLIENTS\_PER\_THREAD = 50;

struct CacheEntry {

std::string response;

};

struct Connection {

int client\_fd;

int server\_fd;

std::string request;

std::string response;

bool headers\_sent;

bool is\_client;

std::string host;

std::string path;

Connection(): client\_fd(-1), server\_fd(-1), headers\_sent(false), is\_client(true) {}

};

struct WorkerData {

std::queue < int > \* client\_queue;

pthread\_mutex\_t \* queue\_mutex;

pthread\_cond\_t \* queue\_cond;

std::map < std::string, CacheEntry > \* cache;

pthread\_mutex\_t \* cache\_mutex;

bool \* running;

};

// Парсинг HTTP-запроса (идентичен заданию 31)

bool parse\_request(const std::string & req, std::string & host, std::string & path) {

size\_t get\_pos = req.find("GET ");

if (get\_pos == std::string::npos) return false;

size\_t http\_pos = req.find(" HTTP/1.");

if (http\_pos == std::string::npos) return false;

std::string url = req.substr(get\_pos + 4, http\_pos - get\_pos - 4);

if (url.substr(0, 7) == "http://") {

url = url.substr(7);

}

size\_t slash\_pos = url.find('/');

if (slash\_pos == std::string::npos) {

host = url;

path = "/";

} else {

host = url.substr(0, slash\_pos);

path = url.substr(slash\_pos);

}

return true;

}

// Подключение к серверу (идентичен заданию 31)

int connect\_to\_server(const std::string & host) {

struct addrinfo hints, \* res;

memset( & hints, 0, sizeof(hints));

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(host.c\_str(), "80", & hints, & res) != 0) {

std::cerr << "getaddrinfo failed: " << gai\_strerror(errno) << std::endl;

return -1;

}

int sock = socket(res -> ai\_family, res -> ai\_socktype, res -> ai\_protocol);

if (sock < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

freeaddrinfo(res);

return -1;

}

if (connect(sock, res -> ai\_addr, res -> ai\_addrlen) < 0) {

std::cerr << "connect failed: " << strerror(errno) << std::endl;

close(sock);

freeaddrinfo(res);

return -1;

}

freeaddrinfo(res);

return sock;

}

// Рабочий поток

void \* worker\_thread(void \* arg) {

WorkerData \* data = (WorkerData \* ) arg;

std::vector < Connection > connections(MAX\_CLIENTS\_PER\_THREAD);

std::vector < struct pollfd > fds;

char buffer[BUFFER\_SIZE];

while ( \* (data -> running)) {

pthread\_mutex\_lock(data -> queue\_mutex);

while (data -> client\_queue -> empty() && \* (data -> running)) {

pthread\_cond\_wait(data -> queue\_cond, data -> queue\_mutex);

}

if (! \* (data -> running) && data -> client\_queue -> empty()) {

pthread\_mutex\_unlock(data -> queue\_mutex);

break;

}

int client\_fd = data -> client\_queue -> front();

data -> client\_queue -> pop();

pthread\_mutex\_unlock(data -> queue\_mutex);

bool found = false;

for (size\_t i = 0; i < connections.size(); ++i) {

if (connections[i].client\_fd == -1) {

connections[i].client\_fd = client\_fd;

connections[i].is\_client = true;

fds.push\_back({

client\_fd,

POLLIN,

0

});

found = true;

break;

}

}

if (!found) {

std::cerr << "Max clients per thread reached" << std::endl;

close(client\_fd);

continue;

}

if (poll(fds.data(), fds.size(), 1000) < 0) {

std::cerr << "poll failed: " << strerror(errno) << std::endl;

continue;

}

for (size\_t i = 0; i < fds.size(); ++i) {

if (fds[i].revents == 0) continue;

int fd = fds[i].fd;

size\_t conn\_idx = 0;

bool is\_client = false;

for (size\_t j = 0; j < connections.size(); ++j) {

if (connections[j].client\_fd == fd) {

conn\_idx = j;

is\_client = true;

break;

} else if (connections[j].server\_fd == fd) {

conn\_idx = j;

break;

}

}

Connection & conn = connections[conn\_idx];

ssize\_t n = read(fd, buffer, BUFFER\_SIZE - 1);

if (n <= 0) {

if (is\_client) {

close(conn.client\_fd);

if (conn.server\_fd >= 0) close(conn.server\_fd);

conn.client\_fd = -1;

conn.server\_fd = -1;

} else {

close(conn.server\_fd);

conn.server\_fd = -1;

}

fds.erase(fds.begin() + i);

--i;

continue;

}

buffer[n] = '\0';

if (is\_client) {

conn.request += buffer;

size\_t header\_end = conn.request.find("\r\n\r\n");

if (header\_end != std::string::npos) {

if (!parse\_request(conn.request, conn.host, conn.path)) {

std::cerr << "Invalid request" << std::endl;

close(conn.client\_fd);

conn.client\_fd = -1;

fds.erase(fds.begin() + i);

--i;

continue;

}

std::string cache\_key = conn.host + conn.path;

pthread\_mutex\_lock(data -> cache\_mutex);

if (data -> cache -> find(cache\_key) != data -> cache -> end()) {

conn.response = ( \* data -> cache)[cache\_key].response;

conn.headers\_sent = true;

write(conn.client\_fd, conn.response.c\_str(), conn.response.size());

close(conn.client\_fd);

conn.client\_fd = -1;

fds.erase(fds.begin() + i);

pthread\_mutex\_unlock(data -> cache\_mutex);

--i;

continue;

}

pthread\_mutex\_unlock(data -> cache\_mutex);

conn.server\_fd = connect\_to\_server(conn.host);

if (conn.server\_fd < 0) {

close(conn.client\_fd);

conn.client\_fd = -1;

fds.erase(fds.begin() + i);

--i;

continue;

}

fds.push\_back({

conn.server\_fd,

POLLIN,

0

});

write(conn.server\_fd, conn.request.c\_str(), conn.request.size());

}

} else {

conn.response += buffer;

if (!conn.headers\_sent) {

size\_t header\_end = conn.response.find("\r\n\r\n");

if (header\_end != std::string::npos) {

conn.headers\_sent = true;

write(conn.client\_fd, conn.response.c\_str(), conn.response.size());

std::string cache\_key = conn.host + conn.path;

pthread\_mutex\_lock(data -> cache\_mutex);

( \* data -> cache)[cache\_key].response = conn.response;

pthread\_mutex\_unlock(data -> cache\_mutex);

}

} else {

write(conn.client\_fd, buffer, n);

std::string cache\_key = conn.host + conn.path;

pthread\_mutex\_lock(data -> cache\_mutex);

( \* data -> cache)[cache\_key].response += buffer;

pthread\_mutex\_unlock(data -> cache\_mutex);

}

}

}

}

for (auto & conn: connections) {

if (conn.client\_fd >= 0) close(conn.client\_fd);

if (conn.server\_fd >= 0) close(conn.server\_fd);

}

return nullptr;

}

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

if (argc != 2) {

std::cerr << "Usage: " << argv[0] << " <num\_threads>" << std::endl;

return 1;

}

int num\_threads = std::atoi(argv[1]);

if (num\_threads <= 0) {

std::cerr << "Invalid number of threads" << std::endl;

return 1;

}

int listen\_fd = socket(AF\_INET, SOCK\_STREAM, 0);

if (listen\_fd < 0) {

std::cerr << "socket failed: " << strerror(errno) << std::endl;

return 1;

}

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_addr.s\_addr = INADDR\_ANY;

addr.sin\_port = htons(LISTEN\_PORT);

int opt = 1;

setsockopt(listen\_fd, SOL\_SOCKET, SO\_REUSEADDR, & opt, sizeof(opt));

if (bind(listen\_fd, (struct sockaddr \* ) & addr, sizeof(addr)) < 0) {

std::cerr << "bind failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

if (listen(listen\_fd, 10) < 0) {

std::cerr << "listen failed: " << strerror(errno) << std::endl;

close(listen\_fd);

return 1;

}

std::cout << "Proxy listening on port " << LISTEN\_PORT << " with " << num\_threads << " worker threads" << std::endl;

std::queue < int > client\_queue;

pthread\_mutex\_t queue\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t queue\_cond = PTHREAD\_COND\_INITIALIZER;

std::map < std::string, CacheEntry > cache;

pthread\_mutex\_t cache\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

bool running = true;

WorkerData data = {

& client\_queue,

& queue\_mutex,

& queue\_cond,

& cache,

& cache\_mutex,

& running

};

std::vector < pthread\_t > threads(num\_threads);

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

if (pthread\_create( & threads[i], nullptr, worker\_thread, & data) != 0) {

std::cerr << "pthread\_create failed: " << strerror(errno) << std::endl;

running = false;

break;

}

}

while (running) {

struct sockaddr\_in client\_addr;

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

int client\_fd = accept(listen\_fd, (struct sockaddr \* ) & client\_addr, & client\_len);

if (client\_fd < 0) {

std::cerr << "accept failed: " << strerror(errno) << std::endl;

continue;

}

pthread\_mutex\_lock( & queue\_mutex);

client\_queue.push(client\_fd);

pthread\_cond\_signal( & queue\_cond);

pthread\_mutex\_unlock( & queue\_mutex);

}

running = false;

pthread\_mutex\_lock( & queue\_mutex);

pthread\_cond\_broadcast( & queue\_cond);

pthread\_mutex\_unlock( & queue\_mutex);

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

pthread\_join(threads[i], nullptr);

}

close(listen\_fd);

pthread\_mutex\_destroy( & queue\_mutex);

pthread\_cond\_destroy( & queue\_cond);

pthread\_mutex\_destroy( & cache\_mutex);

return 0;

}

**Вывод**:

Разработанные программы (задания 28–33) реализуют HTTP-клиент и кэширующий HTTP-прокси, демонстрируя различные подходы к управлению потоками и асинхронной обработкой сетевых соединений в C++ на Linux. Использованные методы — псевдомногопоточность (select, poll, AIO) и многопоточность (отдельные потоки, пул потоков) — подчёркивают баланс между производительностью, масштабируемостью и сложностью реализации. Эти решения иллюстрируют ключевые аспекты оптимизации сетевых приложений.

Каждый подход решает задачу управления множественными соединениями или вводом-выводом, но их выбор зависит от требований к производительности и сложности:

Псевдомногопоточность (select, poll, AIO) идеальна для приложений с ограниченными ресурсами. Она минимизирует накладные расходы, так как работает в одном потоке, избегая синхронизации. select и poll просты и универсальны, но ограничены числом дескрипторов (особенно select). AIO обеспечивает высокую производительность за счёт ядра, но сложнее в реализации и менее переносим.

Многопоточность подходит для масштабируемых приложений. Отдельные потоки (задание 32) упрощают обработку каждого клиента, но создают накладные расходы при большом числе соединений. Пул потоков (задание 33) балансирует производительность и ресурсы, позволяя обрабатывать больше клиентов, чем потоков, с помощью poll в каждом потоке. Два потока в клиенте (задание 30) чётко разделяют сетевые и пользовательские задачи, упрощая логику.

Разработанные решения демонстрируют спектр методов оптимизации потоков в сетевых приложениях — от лёгкой псевдомногопоточности до сложной многопоточности с пулами потоков. Псевдомногопоточность подходит для экономии ресурсов, многопоточность — для масштабируемости и разделения задач. Эти подходы, реализованные в C++ с использованием POSIX API, подчёркивают важность выбора модели потоков в зависимости от требований к производительности, сложности и нагрузке, обеспечивая эффективную работу HTTP-клиентов и прокси.