МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

**Лабораторная работа № 3**

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**:Оптимизация и расширение функциональности потоков в C++

**Ход работы**:

**Задание 11**:Синхронизированный вывод

Модифицируйте программу упр. 1 так, чтобы вывод родительской и дочерней потоках был синхронизирован: сначала родительская поток выводила первую строку, затем дочерняя, затем родительская вторую строку и т.д. Используйте мьютексы. Рекомендуется использовать мьютексы типа РТНREAD\_MUTEX\_ERRORCHECK.

Явные и неявные передачи управления между нитями (sleep(3C)/usleep(3C), sched\_yield(3RT)) и холостые циклы разрешается использовать только на этапе инициализации.

#include <iostream>

#include <pthread.h>

#include <unistd.h> // Для sleep и usleep

#include <sched.h> // Для sched\_yield

#include <time.h> // Для nanosleep

// Глобальные переменные

pthread\_mutex\_t mutex; // Мьютекс для защиты std::cout

pthread\_cond\_t cond; // Условная переменная для синхронизации

bool parent\_turn = true; // Флаг, указывающий, чья очередь (true - родительский поток)

// Функция для дочернего потока

void \* threadFunction(void \* arg) {

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

pthread\_mutex\_lock( & mutex);

// Ждем, пока не наступит очередь дочернего потока

while (parent\_turn) {

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

}

// Выводим строку

std::cout << "Новый поток: строка " << i + 1 << std::endl;

// Переключаем очередь на родительский поток

parent\_turn = true;

pthread\_cond\_signal( & cond); // Сигнализируем родительскому потоку

pthread\_mutex\_unlock( & mutex);

// Используем usleep и sleep для наглядности

usleep(100000); // 100 мс

sleep(1); // 1 секунда

sched\_yield();

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация мьютекса и условной переменной

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutexattr\_init( & mutex\_attr);

// Устанавливаем тип мьютекса PTHREAD\_MUTEX\_ERRORCHECK, как рекомендовано

pthread\_mutexattr\_settype( & mutex\_attr, PTHREAD\_MUTEX\_ERRORCHECK);

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

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

// Создаем поток

if (pthread\_create( & thread, nullptr, threadFunction, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!" << std::endl;

return 1;

}

// Родительский поток

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

pthread\_mutex\_lock( & mutex);

// Ждем, пока не наступит очередь родительского потока

while (!parent\_turn) {

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

}

// Выводим строку

std::cout << "Родительский поток: строка " << i + 1 << std::endl;

// Переключаем очередь на дочерний поток

parent\_turn = false;

pthread\_cond\_signal( & cond); // Сигнализируем дочернему потоку

pthread\_mutex\_unlock( & mutex);

// Используем nanosleep и sleep для наглядности

struct timespec ts;

ts.tv\_sec = 0;

ts.tv\_nsec = 100000000; // 100 мс

nanosleep( & ts, nullptr);

sleep(1); // 1 секунда

sched\_yield();

}

// Ожидаем завершения дочернего потока

pthread\_join(thread, nullptr);

// Уничтожаем мьютекс, условную переменную и атрибуты

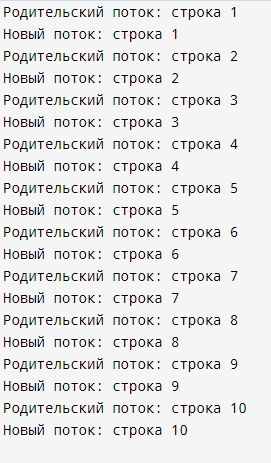
pthread\_mutex\_destroy( & mutex);

pthread\_cond\_destroy( & cond);

pthread\_mutexattr\_destroy( & mutex\_attr);

return 0;

}



**Задание 12**:Синхронизация вывода 2

**Условие**: Докажите, что задача 11 не может быть решена с использованием двух мьютексов без использования других средств синхронизации.

**Ответ**: Использование только двух мьютексов (без условных переменных или других механизмов синхронизации) не позволяет гарантировать строгое чередование между потоками. Вот почему:

1. **Роль мьютексов**: Мьютексы предназначены для защиты критических секций, чтобы предотвратить одновременный доступ к общим ресурсам (например, std::cout). Один мьютекс может защитить вывод, чтобы строки не перемешивались, но он не может управлять порядком выполнения потоков.
2. **Попытка с двумя мьютексами**:
   * Предположим, у нас есть два мьютекса: mutex1 и mutex2.
   * Родительский поток захватывает mutex1, выводит строку, затем освобождает mutex1 и пытается захватить mutex2.
   * Дочерний поток захватывает mutex2, выводит строку, затем освобождает mutex2 и пытается захватить mutex1.
   * Проблема: мьютексы не могут гарантировать, что после освобождения mutex1 родительским потоком дочерний поток сразу захватит его. Планировщик потоков может снова дать управление родительскому потоку, и он снова захватит mutex1, нарушив чередование.
3. **Отсутствие механизма ожидания**: Мьютексы не предоставляют способа "ждать" своей очереди. Без условных переменных (pthread\_cond\_t) или других средств (например, семафоров) потоки не могут синхронизироваться так, чтобы гарантировать чередование. Один поток может захватывать мьютекс многократно, пока другой ждет, что приводит к выводу всех строк одного потока подряд.
4. **Вывод**: Для строгого чередования нужно средство, которое позволяет потоку ждать, пока другой поток не завершит свою работу и не "разрешит" ему продолжить. Мьютексы сами по себе этого не обеспечивают. В задаче 11 мы использовали условные переменные (pthread\_cond\_t) и флаг parent\_turn, чтобы добиться чередования, что подтверждает, что только мьютексы недостаточны.

**Задание 13**:Синхронизация вывода 3

**Условие**: Решите задачу 11 с использованием условной переменной и минимально необходимого количества мьютексов.

**Решение**: Мы уже решили задачу 11 с использованием одного мьютекса и одной условной переменной, что является минимально необходимым количеством. Один мьютекс защищает доступ к std::cout и используется для работы с условной переменной, а условная переменная управляет чередованием. Код из предыдущего ответа уже соответствует этому требованию:

* Один мьютекс (pthread\_mutex\_t mutex) защищает критическую секцию (std::cout) и используется с условной переменной.
* Одна условная переменная (pthread\_cond\_t cond) и флаг (parent\_turn) управляют чередованием.

Минимально необходимое количество мьютексов — **один**, так как нам нужно только защитить общий ресурс (std::cout) и синхронизировать ожидание с помощью условной переменной. Два мьютекса не нужны, так как чередование обеспечивается условной переменной.

Код остается тем же, что был в предыдущем ответе (см. решение задачи 11).

**Задание 14**:Синхронизация вывода 4

**Условие**: Решите задачу 11 с использованием двух семафоров-счетчиков:

#include <iostream>

#include <pthread.h>

#include <semaphore.h> // Для семафоров

#include <unistd.h>

#include <sched.h>

#include <time.h>

// Глобальные семафоры

sem\_t sem\_parent; // Семафор для родительского потока

sem\_t sem\_child; // Семафор для дочернего потока

// Функция для дочернего потока

void \* threadFunction(void \* arg) {

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

// Ждем, пока родительский поток не завершит свою итерацию

sem\_wait( & sem\_child);

// Выводим строку

std::cout << "Новый поток: строка " << i + 1 << std::endl;

// Разрешаем родительскому потоку продолжить

sem\_post( & sem\_parent);

usleep(100000); // 100 мс для наглядности

sched\_yield();

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация семафоров

sem\_init( & sem\_parent, 0, 1); // Родительский поток начинает (1)

sem\_init( & sem\_child, 0, 0); // Дочерний поток ждет (0)

// Создаем поток

if (pthread\_create( & thread, nullptr, threadFunction, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!" << std::endl;

return 1;

}

// Родительский поток

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

// Ждем своей очереди

sem\_wait( & sem\_parent);

// Выводим строку

std::cout << "Родительский поток: строка " << i + 1 << std::endl;

// Разрешаем дочернему потоку продолжить

sem\_post( & sem\_child);

// Задержка с помощью nanosleep

struct timespec ts;

ts.tv\_sec = 0;

ts.tv\_nsec = 100000000; // 100 мс

nanosleep( & ts, nullptr);

sched\_yield();

}

// Ожидаем завершения дочернего потока

pthread\_join(thread, nullptr);

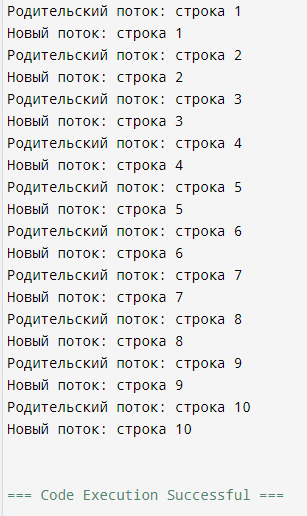
// Уничтожаем семафоры

sem\_destroy( & sem\_parent);

sem\_destroy( & sem\_child);

return 0;

}



**Задание 15**:Синхронизация вывода 5

**Условие**: Если вы решили задачи 12 и 14, объясните, почему ваше доказательство неприменимо к семафор-счетчикам.

**Ответ**: В задаче 12 мы доказали, что два мьютекса без других средств синхронизации не могут обеспечить строгое чередование, потому что мьютексы не предоставляют механизма ожидания очереди: они только защищают критические секции, но не управляют порядком выполнения потоков.

Это доказательство **неприменимо** к семафор-счетчикам по следующим причинам:

1. **Семафоры имеют встроенный механизм ожидания и управления очередностью**:
   * Семафоры (sem\_t) поддерживают счетчик, который позволяет "разрешать" или "запрещать" выполнение потока. Когда счетчик равен 0, поток, вызывающий sem\_wait, блокируется, пока другой поток не вызовет sem\_post и не увеличит счетчик.
   * В задаче 14 мы использовали два семафора: sem\_parent и sem\_child. Один поток ждет, пока другой не завершит свою итерацию и не вызовет sem\_post, что обеспечивает строгое чередование.
2. **Мьютексы не могут "сигнализировать"**:
   * Мьютексы предназначены только для защиты ресурсов. Если поток освобождает мьютекс, нет гарантии, что другой поток сразу захватит его — планировщик может снова дать управление первому потоку.
   * Семафоры, напротив, позволяют одному потоку "сигнализировать" другому, увеличивая счетчик и разблокируя ожидающий поток.
3. **Семафоры решают проблему чередования**:
   * В задаче 14 мы инициализировали sem\_parent значением 1 (родительский поток начинает), а sem\_child — значением 0 (дочерний поток ждет). Это гарантирует, что потоки будут чередоваться, так как каждый поток ждет своей очереди, пока другой не завершит свою работу.

**Вывод**: Доказательство из задачи 12 неприменимо к семафорам, потому что семафоры предоставляют дополнительный механизм синхронизации (счетчик и ожидание), которого нет у мьютексов. Семафоры позволяют управлять очередностью выполнения потоков, что делает их подходящими для задачи строгого чередования.

**Задание 16**:Синхронизированный вывод двух процессов

**Условие**: Решите задачу 11 с использованием двух процессов (а не потоков) и именованных семафоров-счетчиков:

#include <iostream>

#include <semaphore.h>

#include <fcntl.h> // Для O\_CREAT

#include <sys/stat.h> // Для S\_IRUSR, S\_IWUSR

#include <unistd.h> // Для fork, sleep, usleep

#include <sched.h>

#include <time.h>

#include <sys/wait.h> // Для wait

int main() {

// Именованные семафоры

sem\_t \* sem\_parent = sem\_open("/sem\_parent", O\_CREAT, S\_IRUSR | S\_IWUSR, 1); // Родительский процесс начинает

sem\_t \* sem\_child = sem\_open("/sem\_child", O\_CREAT, S\_IRUSR | S\_IWUSR, 0); // Дочерний процесс ждет

if (sem\_parent == SEM\_FAILED || sem\_child == SEM\_FAILED) {

std::cerr << "Ошибка при создании семафоров!" << std::endl;

return 1;

}

// Создаем дочерний процесс

pid\_t pid = fork();

if (pid < 0) {

std::cerr << "Ошибка при создании процесса!" << std::endl;

return 1;

}

if (pid == 0) { // Дочерний процесс

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

sem\_wait(sem\_child); // Ждем своей очереди

// Выводим строку

std::cout << "Новый процесс: строка " << i + 1 << std::endl;

sem\_post(sem\_parent); // Разрешаем родительскому процессу продолжить

usleep(100000); // 100 мс

sched\_yield();

}

// Закрываем семафоры в дочернем процессе

sem\_close(sem\_parent);

sem\_close(sem\_child);

} else { // Родительский процесс

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

sem\_wait(sem\_parent); // Ждем своей очереди

// Выводим строку

std::cout << "Родительский процесс: строка " << i + 1 << std::endl;

sem\_post(sem\_child); // Разрешаем дочернему процессу продолжить

struct timespec ts;

ts.tv\_sec = 0;

ts.tv\_nsec = 100000000; // 100 мс

nanosleep( & ts, nullptr);

sched\_yield();

}

// Ожидаем завершения дочернего процесса

wait(nullptr);

// Закрываем и удаляем семафоры

sem\_close(sem\_parent);

sem\_close(sem\_child);

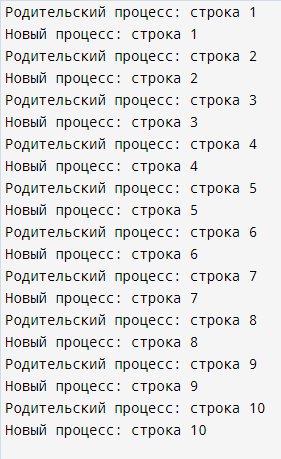
sem\_unlink("/sem\_parent");

sem\_unlink("/sem\_child");

}

return 0;

}



**Задание 17**:Синхронизированный доступ к списку

**Условие**: Родительская поток программы должна считывать вводимые пользователем строки и помещать их в начало связанного списка. Строки длиннее 80 символов можно разрезать на несколько строк. При вводе пустой строки программа должна выдавать текущее состояние списка. Дочерняя поток пробуждается каждые пять секунд и сортирует список в лексикографическом порядке (используйте пузырьковую сортировку). Все операции над списком должны синхронизироваться при помощи мьютекса:

#include <iostream>

#include <pthread.h>

#include <unistd.h> // Для sleep

#include <string>

#include <list> // Для связанного списка

// Глобальные переменные

pthread\_mutex\_t mutex; // Мьютекс для защиты списка

std::list < std::string > string\_list; // Связанный список для хранения строк

// Функция для разделения строки на части по 80 символов

void split\_string(const std::string & input) {

size\_t len = input.length();

size\_t pos = 0;

while (pos < len) {

size\_t chunk\_size = std::min < size\_t > (80, len - pos);

std::string chunk = input.substr(pos, chunk\_size);

pthread\_mutex\_lock( & mutex);

string\_list.push\_front(chunk); // Добавляем в начало списка

pthread\_mutex\_unlock( & mutex);

pos += chunk\_size;

}

}

// Функция для вывода списка

void print\_list() {

pthread\_mutex\_lock( & mutex);

if (string\_list.empty()) {

std::cout << "Список пуст\n";

} else {

std::cout << "Текущее состояние списка:\n";

int index = 1;

for (const auto & str: string\_list) {

std::cout << index++ << ". " << str << "\n";

}

}

pthread\_mutex\_unlock( & mutex);

}

// Функция пузырьковой сортировки для std::list (по возрастанию)

void bubble\_sort(std::list < std::string > & lst) {

if (lst.empty()) return;

bool swapped;

do {

swapped = false;

auto it = lst.begin();

auto next\_it = std::next(it);

// Проходим по списку, сравнивая соседние элементы

while (next\_it != lst.end()) {

if ( \* it > \* next\_it) { // Сортировка по возрастанию (a < b)

std::swap( \* it, \* next\_it);

swapped = true;

}

++it;

++next\_it;

}

} while (swapped); // Повторяем, пока есть обмены

}

// Функция для дочернего потока (сортировка каждые 5 секунд)

void \* sortThread(void \* arg) {

while (true) {

sleep(5); // Ждем 5 секунд

pthread\_mutex\_lock( & mutex);

if (!string\_list.empty()) {

bubble\_sort(string\_list); // Используем пузырьковую сортировку

std::cout << "Дочерний поток: список отсортирован (пузырьковая сортировка)\n";

}

pthread\_mutex\_unlock( & mutex);

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация мьютекса

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutexattr\_init( & mutex\_attr);

pthread\_mutexattr\_settype( & mutex\_attr, PTHREAD\_MUTEX\_ERRORCHECK); // Используем ERRORCHECK для отладки

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

// Создаем дочерний поток для сортировки

if (pthread\_create( & thread, nullptr, sortThread, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!\n";

return 1;

}

// Родительский поток: ввод строк

std::string input;

std::cout << "Введите строки (пустая строка для вывода списка, Ctrl+D для выхода):\n";

while (std::getline(std::cin, input)) {

if (input.empty()) {

// Пустая строка — выводим список

print\_list();

} else {

// Разделяем строку, если она длиннее 80 символов, и добавляем в список

split\_string(input);

}

std::cout << "Введите следующую строку:\n";

}

// Ожидаем завершения дочернего потока

pthread\_cancel(thread); // Принудительно завершаем поток (для примера)

pthread\_join(thread, nullptr);

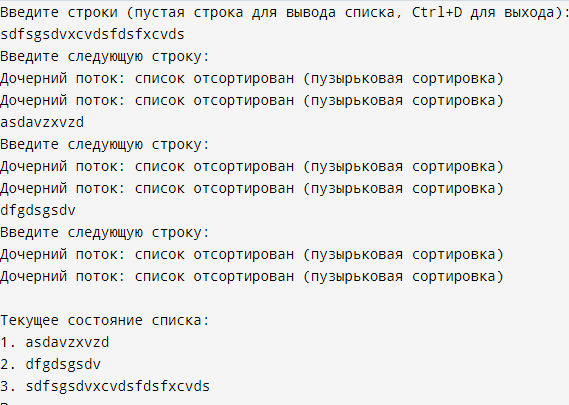
// Уничтожаем мьютекс и атрибуты

pthread\_mutex\_destroy( & mutex);

pthread\_mutexattr\_destroy( & mutex\_attr);

return 0;

}



**Задание 18**:Синхронизированный доступ к списку 2

**Условие**: Переделайте программу упр. 17 так, чтобы с каждой записью (а также с заголовком списка) был связан свой собственный мьютекс.

Примечание: при перестановке записей списка, необходимой при реализации пузырьковой сортировки, необходимо блокировать мьютексы трех записей.

Примечание 2: чтобы избежать мертвых блокировок, мьютексы записей, более близких к началу списка, всегда захватывайте раньше.

Примечание 3: преподаватель может потребовать, чтобы программа включала две или более сортирующие нити, а также потребовать изменить интервал между сортировками.

#include <iostream>

#include <pthread.h>

#include <unistd.h> // Для sleep

#include <string>

#include <list> // Для связанного списка

// Структура для записи в списке (строка + мьютекс)

struct ListNode {

std::string data;

pthread\_mutex\_t mutex;

ListNode(const std::string & str): data(str) {

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutexattr\_init( & mutex\_attr);

pthread\_mutexattr\_settype( & mutex\_attr, PTHREAD\_MUTEX\_ERRORCHECK); // Для отладки

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

pthread\_mutexattr\_destroy( & mutex\_attr);

}

~ListNode() {

pthread\_mutex\_destroy( & mutex);

}

};

// Глобальные переменные

std::list < ListNode > string\_list; // Связанный список с мьютексами для каждой записи

pthread\_mutex\_t list\_mutex; // Мьютекс для заголовка списка (для операций добавления/удаления)

// Функция для разделения строки на части по 80 символов

void split\_string(const std::string & input) {

size\_t len = input.length();

size\_t pos = 0;

while (pos < len) {

size\_t chunk\_size = std::min < size\_t > (80, len - pos);

std::string chunk = input.substr(pos, chunk\_size);

pthread\_mutex\_lock( & list\_mutex); // Защищаем добавление в список

string\_list.push\_front(ListNode(chunk)); // Добавляем в начало списка

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

pos += chunk\_size;

}

}

// Функция для вывода списка

void print\_list() {

pthread\_mutex\_lock( & list\_mutex); // Защищаем доступ к списку

if (string\_list.empty()) {

std::cout << "Список пуст\n";

} else {

std::cout << "Текущее состояние списка:\n";

int index = 1;

for (auto & node: string\_list) {

pthread\_mutex\_lock( & node.mutex); // Защищаем доступ к данным узла

std::cout << index++ << ". " << node.data << "\n";

pthread\_mutex\_unlock( & node.mutex);

}

}

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

}

// Функция пузырьковой сортировки с мьютексами

void bubble\_sort(std::list < ListNode > & lst) {

if (lst.empty()) return;

bool swapped;

do {

swapped = false;

auto it = lst.begin();

auto next\_it = std::next(it);

while (next\_it != lst.end()) {

auto prev\_it = it;

if (it == lst.begin()) {

// Если текущий элемент — первый, предыдущего нет

prev\_it = lst.end();

} else {

prev\_it = std::prev(it);

}

// Захватываем мьютексы в порядке от начала к концу списка

if (prev\_it != lst.end()) {

pthread\_mutex\_lock( & prev\_it -> mutex); // Предыдущий узел

}

pthread\_mutex\_lock( & it -> mutex); // Текущий узел

pthread\_mutex\_lock( & next\_it -> mutex); // Следующий узел

// Сравниваем строки

if (it -> data > next\_it -> data) {

// Меняем данные местами

std::swap(it -> data, next\_it -> data);

swapped = true;

}

// Освобождаем мьютексы в обратном порядке

pthread\_mutex\_unlock( & next\_it -> mutex);

pthread\_mutex\_unlock( & it -> mutex);

if (prev\_it != lst.end()) {

pthread\_mutex\_unlock( & prev\_it -> mutex);

}

++it;

++next\_it;

}

} while (swapped); // Повторяем, пока есть обмены

}

// Функция для дочернего потока (сортировка каждые 5 секунд)

void \* sortThread(void \* arg) {

while (true) {

sleep(5); // Ждем 5 секунд

pthread\_mutex\_lock( & list\_mutex); // Защищаем доступ к списку

if (!string\_list.empty()) {

bubble\_sort(string\_list); // Используем пузырьковую сортировку

std::cout << "Дочерний поток: список отсортирован (пузырьковая сортировка)\n";

}

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

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация мьютекса для заголовка списка

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutexattr\_init( & mutex\_attr);

pthread\_mutexattr\_settype( & mutex\_attr, PTHREAD\_MUTEX\_ERRORCHECK);

pthread\_mutex\_init( & list\_mutex, & mutex\_attr);

// Создаем дочерний поток для сортировки

if (pthread\_create( & thread, nullptr, sortThread, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!\n";

return 1;

}

// Родительский поток: ввод строк

std::string input;

std::cout << "Введите строки (пустая строка для вывода списка, Ctrl+D для выхода):\n";

while (std::getline(std::cin, input)) {

if (input.empty()) {

// Пустая строка — выводим список

print\_list();

} else {

// Разделяем строку, если она длиннее 80 символов, и добавляем в список

split\_string(input);

}

std::cout << "Введите следующую строку:\n";

}

// Ожидаем завершения дочернего потока

pthread\_cancel(thread); // Принудительно завершаем поток (для примера)

pthread\_join(thread, nullptr);

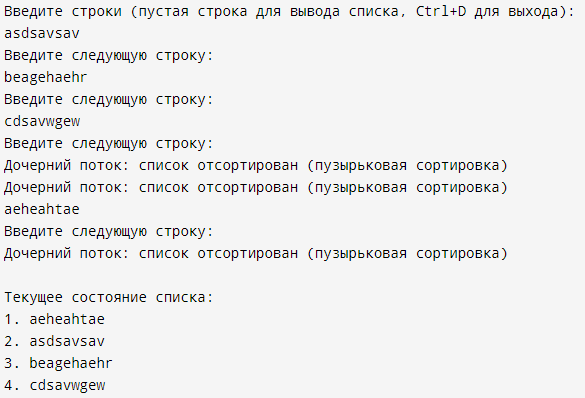
// Уничтожаем мьютекс и атрибуты

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

pthread\_mutexattr\_destroy( & mutex\_attr);

return 0;

}



**Задание 19**:Синхронизированный доступ к списку 3

**Условие**: Модифицируйте программу упр. 18 так, чтобы дочерняя поток засыпала на одну секунду между исполнениями каждого шага сортировки (между перестановками записей в списке). При этом можно будет наблюдать процесс сортировки по шагам.

#include <iostream>

#include <pthread.h>

#include <unistd.h> // Для sleep

#include <string>

#include <list> // Для связанного списка

// Структура для записи в списке (строка + мьютекс)

struct ListNode {

std::string data;

pthread\_mutex\_t mutex;

ListNode(const std::string & str): data(str) {

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutexattr\_init( & mutex\_attr);

pthread\_mutexattr\_settype( & mutex\_attr, PTHREAD\_MUTEX\_ERRORCHECK);

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

pthread\_mutexattr\_destroy( & mutex\_attr);

}

~ListNode() {

pthread\_mutex\_destroy( & mutex);

}

};

// Глобальные переменные

std::list < ListNode > string\_list; // Связанный список с мьютексами для каждой записи

pthread\_mutex\_t list\_mutex; // Мьютекс для заголовка списка

// Функция для разделения строки на части по 80 символов

void split\_string(const std::string & input) {

size\_t len = input.length();

size\_t pos = 0;

while (pos < len) {

size\_t chunk\_size = std::min < size\_t > (80, len - pos);

std::string chunk = input.substr(pos, chunk\_size);

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

string\_list.push\_front(ListNode(chunk));

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

pos += chunk\_size;

}

}

// Функция для вывода списка

void print\_list() {

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

if (string\_list.empty()) {

std::cout << "Список пуст\n";

} else {

std::cout << "Текущее состояние списка:\n";

int index = 1;

for (auto & node: string\_list) {

pthread\_mutex\_lock( & node.mutex);

std::cout << index++ << ". " << node.data << "\n";

pthread\_mutex\_unlock( & node.mutex);

}

}

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

}

// Функция пузырьковой сортировки с мьютексами и задержкой

void bubble\_sort(std::list < ListNode > & lst) {

if (lst.empty()) return;

bool swapped;

do {

swapped = false;

auto it = lst.begin();

auto next\_it = std::next(it);

while (next\_it != lst.end()) {

auto prev\_it = it;

if (it == lst.begin()) {

prev\_it = lst.end();

} else {

prev\_it = std::prev(it);

}

// Захватываем мьютексы в порядке от начала к концу списка

if (prev\_it != lst.end()) {

pthread\_mutex\_lock( & prev\_it -> mutex);

}

pthread\_mutex\_lock( & it -> mutex);

pthread\_mutex\_lock( & next\_it -> mutex);

// Сравниваем строки

if (it -> data > next\_it -> data) {

// Меняем данные местами

std::swap(it -> data, next\_it -> data);

swapped = true;

// Задержка 1 секунда после перестановки

std::cout << "Перестановка: " << it -> data << " и " << next\_it -> data << "\n";

sleep(1); // Задержка для наблюдения

}

// Освобождаем мьютексы

pthread\_mutex\_unlock( & next\_it -> mutex);

pthread\_mutex\_unlock( & it -> mutex);

if (prev\_it != lst.end()) {

pthread\_mutex\_unlock( & prev\_it -> mutex);

}

++it;

++next\_it;

}

} while (swapped);

}

// Функция для дочернего потока (сортировка каждые 5 секунд)

void \* sortThread(void \* arg) {

while (true) {

sleep(5); // Ждем 5 секунд

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

if (!string\_list.empty()) {

std::cout << "Дочерний поток: начинаю сортировку\n";

bubble\_sort(string\_list);

std::cout << "Дочерний поток: список отсортирован (пузырьковая сортировка)\n";

}

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

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация мьютекса для заголовка списка

pthread\_mutexattr\_t mutex\_attr;

pthread\_mutexattr\_init( & mutex\_attr);

pthread\_mutexattr\_settype( & mutex\_attr, PTHREAD\_MUTEX\_ERRORCHECK);

pthread\_mutex\_init( & list\_mutex, & mutex\_attr);

// Создаем дочерний поток для сортировки

if (pthread\_create( & thread, nullptr, sortThread, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!\n";

return 1;

}

// Родительский поток: ввод строк

std::string input;

std::cout << "Введите строки (пустая строка для вывода списка, Ctrl+D для выхода):\n";

while (std::getline(std::cin, input)) {

if (input.empty()) {

print\_list();

} else {

split\_string(input);

}

std::cout << "Введите следующую строку:\n";

}

// Ожидаем завершения дочернего потока

pthread\_cancel(thread);

pthread\_join(thread, nullptr);

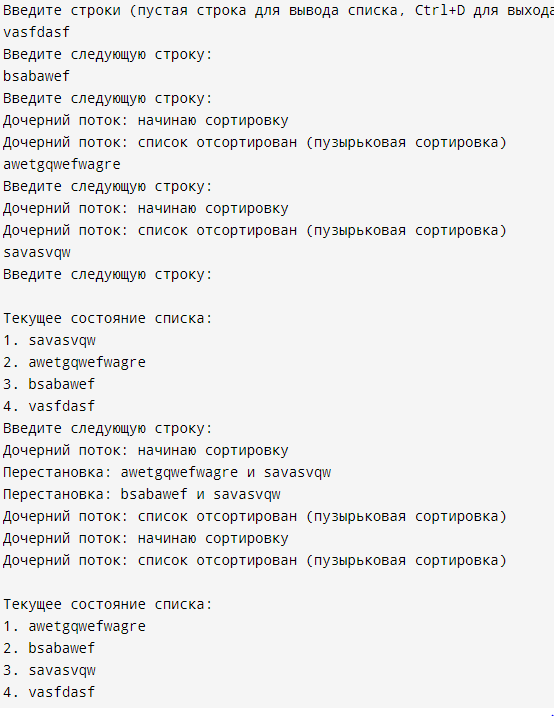
// Уничтожаем мьютекс и атрибуты

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

pthread\_mutexattr\_destroy( & mutex\_attr);

return 0;

}



**Задание 20**:Использование блокировки чтения-записи

**Условие**: Модифицируйте программу упр. 18 так, чтобы вместо мьютекса использовалась блокировка чтения-записи.

#include <iostream>

#include <pthread.h>

#include <unistd.h> // Для sleep

#include <string>

#include <list> // Для связанного списка

// Структура для записи в списке (строка + read-write lock)

struct ListNode {

std::string data;

pthread\_rwlock\_t rwlock;

ListNode(const std::string & str): data(str) {

pthread\_rwlock\_init( & rwlock, nullptr);

}

~ListNode() {

pthread\_rwlock\_destroy( & rwlock);

}

};

// Глобальные переменные

std::list < ListNode > string\_list; // Связанный список с rwlock для каждой записи

pthread\_rwlock\_t list\_rwlock; // Read-write lock для заголовка списка

// Функция для разделения строки на части по 80 символов

void split\_string(const std::string & input) {

size\_t len = input.length();

size\_t pos = 0;

while (pos < len) {

size\_t chunk\_size = std::min < size\_t > (80, len - pos);

std::string chunk = input.substr(pos, chunk\_size);

pthread\_rwlock\_wrlock( & list\_rwlock); // Блокируем запись

string\_list.push\_front(ListNode(chunk));

pthread\_rwlock\_unlock( & list\_rwlock);

pos += chunk\_size;

}

}

// Функция для вывода списка (чтение)

void print\_list() {

pthread\_rwlock\_rdlock( & list\_rwlock); // Блокируем только для чтения

if (string\_list.empty()) {

std::cout << "Список пуст\n";

} else {

std::cout << "Текущее состояние списка:\n";

int index = 1;

for (auto & node: string\_list) {

pthread\_rwlock\_rdlock( & node.rwlock); // Чтение данных узла

std::cout << index++ << ". " << node.data << "\n";

pthread\_rwlock\_unlock( & node.rwlock);

}

}

pthread\_rwlock\_unlock( & list\_rwlock);

}

// Функция пузырьковой сортировки с rwlock

void bubble\_sort(std::list < ListNode > & lst) {

if (lst.empty()) return;

bool swapped;

do {

swapped = false;

auto it = lst.begin();

auto next\_it = std::next(it);

while (next\_it != lst.end()) {

auto prev\_it = it;

if (it == lst.begin()) {

prev\_it = lst.end();

} else {

prev\_it = std::prev(it);

}

// Захватываем rwlock в режиме записи (нужна эксклюзивная блокировка)

if (prev\_it != lst.end()) {

pthread\_rwlock\_wrlock( & prev\_it -> rwlock);

}

pthread\_rwlock\_wrlock( & it -> rwlock);

pthread\_rwlock\_wrlock( & next\_it -> rwlock);

// Сравниваем строки

if (it -> data > next\_it -> data) {

std::swap(it -> data, next\_it -> data);

swapped = true;

}

// Освобождаем rwlock

pthread\_rwlock\_unlock( & next\_it -> rwlock);

pthread\_rwlock\_unlock( & it -> rwlock);

if (prev\_it != lst.end()) {

pthread\_rwlock\_unlock( & prev\_it -> rwlock);

}

++it;

++next\_it;

}

} while (swapped);

}

// Функция для дочернего потока (сортировка каждые 5 секунд)

void \* sortThread(void \* arg) {

while (true) {

sleep(5); // Ждем 5 секунд

pthread\_rwlock\_wrlock( & list\_rwlock); // Блокируем запись

if (!string\_list.empty()) {

bubble\_sort(string\_list);

std::cout << "Дочерний поток: список отсортирован (пузырьковая сортировка)\n";

}

pthread\_rwlock\_unlock( & list\_rwlock);

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация rwlock для заголовка списка

pthread\_rwlock\_init( & list\_rwlock, nullptr);

// Создаем дочерний поток для сортировки

if (pthread\_create( & thread, nullptr, sortThread, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!\n";

return 1;

}

// Родительский поток: ввод строк

std::string input;

std::cout << "Введите строки (пустая строка для вывода списка, Ctrl+D для выхода):\n";

while (std::getline(std::cin, input)) {

if (input.empty()) {

print\_list();

} else {

split\_string(input);

}

std::cout << "Введите следующую строку:\n";

}

// Ожидаем завершения дочернего потока

pthread\_cancel(thread);

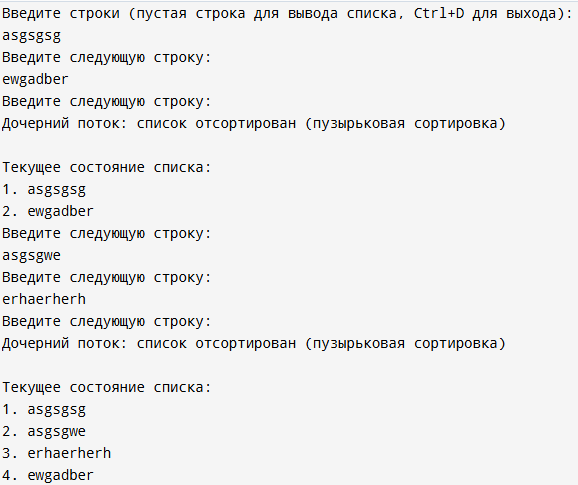
pthread\_join(thread, nullptr);

// Уничтожаем rwlock

pthread\_rwlock\_destroy( & list\_rwlock);

return 0;

}



**Задание 21**:Использование блокировки чтения-записи 2

**Условие**: Модифицируйте аналогичным образом программу упр. 19.

#include <iostream>

#include <pthread.h>

#include <unistd.h> // Для sleep

#include <string>

#include <list> // Для связанного списка

// Структура для записи в списке (строка + read-write lock)

struct ListNode {

std::string data;

pthread\_rwlock\_t rwlock;

ListNode(const std::string & str): data(str) {

pthread\_rwlock\_init( & rwlock, nullptr);

}

~ListNode() {

pthread\_rwlock\_destroy( & rwlock);

}

};

// Глобальные переменные

std::list < ListNode > string\_list; // Связанный список с rwlock для каждой записи

pthread\_rwlock\_t list\_rwlock; // Read-write lock для заголовка списка

// Функция для разделения строки на части по 80 символов

void split\_string(const std::string & input) {

size\_t len = input.length();

size\_t pos = 0;

while (pos < len) {

size\_t chunk\_size = std::min < size\_t > (80, len - pos);

std::string chunk = input.substr(pos, chunk\_size);

pthread\_rwlock\_wrlock( & list\_rwlock); // Блокируем запись

string\_list.push\_front(ListNode(chunk));

pthread\_rwlock\_unlock( & list\_rwlock);

pos += chunk\_size;

}

}

// Функция для вывода списка (чтение)

void print\_list() {

pthread\_rwlock\_rdlock( & list\_rwlock); // Блокируем только для чтения

if (string\_list.empty()) {

std::cout << "Список пуст\n";

} else {

std::cout << "Текущее состояние списка:\n";

int index = 1;

for (auto & node: string\_list) {

pthread\_rwlock\_rdlock( & node.rwlock); // Чтение данных узла

std::cout << index++ << ". " << node.data << "\n";

pthread\_rwlock\_unlock( & node.rwlock);

}

}

pthread\_rwlock\_unlock( & list\_rwlock);

}

// Функция пузырьковой сортировки с rwlock и задержкой

void bubble\_sort(std::list < ListNode > & lst) {

if (lst.empty()) return;

bool swapped;

do {

swapped = false;

auto it = lst.begin();

auto next\_it = std::next(it);

while (next\_it != lst.end()) {

auto prev\_it = it;

if (it == lst.begin()) {

prev\_it = lst.end();

} else {

prev\_it = std::prev(it);

}

// Захватываем rwlock в режиме записи

if (prev\_it != lst.end()) {

pthread\_rwlock\_wrlock( & prev\_it -> rwlock);

}

pthread\_rwlock\_wrlock( & it -> rwlock);

pthread\_rwlock\_wrlock( & next\_it -> rwlock);

// Сравниваем строки

if (it -> data > next\_it -> data) {

std::swap(it -> data, next\_it -> data);

swapped = true;

// Задержка 1 секунда после перестановки

std::cout << "Перестановка: " << it -> data << " и " << next\_it -> data << "\n";

sleep(1); // Задержка для наблюдения

}

// Освобождаем rwlock

pthread\_rwlock\_unlock( & next\_it -> rwlock);

pthread\_rwlock\_unlock( & it -> rwlock);

if (prev\_it != lst.end()) {

pthread\_rwlock\_unlock( & prev\_it -> rwlock);

}

++it;

++next\_it;

}

} while (swapped);

}

// Функция для дочернего потока (сортировка каждые 5 секунд)

void \* sortThread(void \* arg) {

while (true) {

sleep(5); // Ждем 5 секунд

pthread\_rwlock\_wrlock( & list\_rwlock); // Блокируем запись

if (!string\_list.empty()) {

std::cout << "Дочерний поток: начинаю сортировку\n";

bubble\_sort(string\_list);

std::cout << "Дочерний поток: список отсортирован (пузырьковая сортировка)\n";

}

pthread\_rwlock\_unlock( & list\_rwlock);

}

return nullptr;

}

int main() {

pthread\_t thread;

// Инициализация rwlock для заголовка списка

pthread\_rwlock\_init( & list\_rwlock, nullptr);

// Создаем дочерний поток для сортировки

if (pthread\_create( & thread, nullptr, sortThread, nullptr) != 0) {

std::cerr << "Ошибка при создании потока!\n";

return 1;

}

// Родительский поток: ввод строк

std::string input;

std::cout << "Введите строки (пустая строка для вывода списка, Ctrl+D для выхода):\n";

while (std::getline(std::cin, input)) {

if (input.empty()) {

print\_list();

} else {

split\_string(input);

}

std::cout << "Введите следующую строку:\n";

}

// Ожидаем завершения дочернего потока

pthread\_cancel(thread);

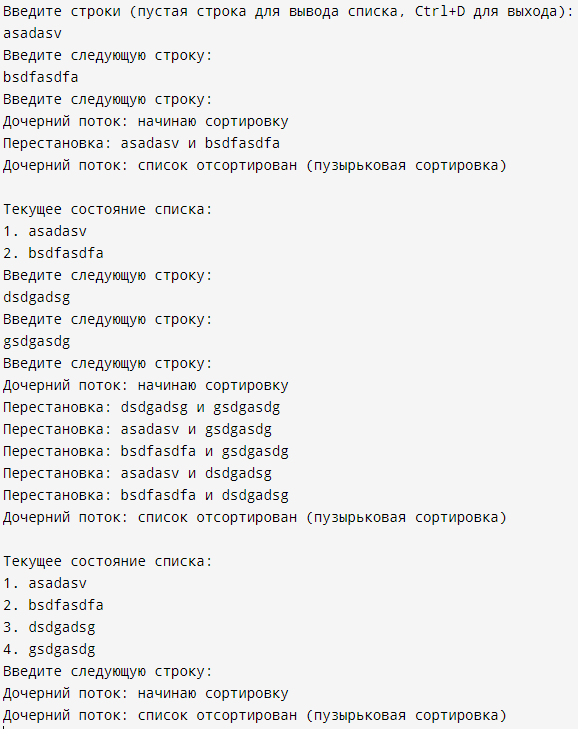
pthread\_join(thread, nullptr);

// Уничтожаем rwlock

pthread\_rwlock\_destroy( & list\_rwlock);

return 0;

}



**Задание 22**:Обедающие философы 2

**Условие**: Решите задачу упр. 10 при помощи атомарного захвата вилок. Когда философ может взять одну вилку, но не может взять другую, он должен положить вилку на стол и ждать, пока освободятся обе вилки.

Рекомендация: создайте еще мьютекс forks и условную переменную. При попытке взять вилку философ должен захватывать forks и проверять доступность обеих вилок при помощи pthread\_trylock(ЗС). Если одна из вилок недоступна, философ должен освободить вторую вилку (если он успел ее захватить) и заснуть на условной переменной. Освобождая вилки, философ должен оповещать остальных философов об этом при помощи условной переменной. Тщательно продумайте процедуру захвата и освобождения мьютексов, чтобы избежать ошибок потерянного пробуждения.

#include <iostream>

#include <pthread.h>

#include <unistd.h>

#include <cstdlib>

const int PHILO = 5;

const int DELAY = 30000;

const int FOOD = 50;

pthread\_mutex\_t forks[PHILO];

pthread\_t phils[PHILO];

pthread\_mutex\_t foodlock;

pthread\_mutex\_t cout\_lock; // Мьютекс для синхронизации вывода

pthread\_mutex\_t forks\_lock; // Мьютекс для синхронизации доступа к вилкам

pthread\_cond\_t cond; // Условная переменная для ожидания вилок

int sleep\_seconds = 0;

void \* philosopher(void \* id);

int food\_on\_table();

void get\_forks(int phil, int fork1, int fork2);

void down\_forks(int fork1, int fork2);

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

if (argc == 2) {

sleep\_seconds = std::atoi(argv[1]);

}

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

pthread\_mutex\_init( & cout\_lock, nullptr);

pthread\_mutex\_init( & forks\_lock, nullptr);

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

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

pthread\_mutex\_init( & forks[i], nullptr);

}

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

pthread\_create( & phils[i], nullptr, philosopher, (void \* )(long) i);

}

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

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

}

pthread\_mutex\_destroy( & foodlock);

pthread\_mutex\_destroy( & cout\_lock);

pthread\_mutex\_destroy( & forks\_lock);

pthread\_cond\_destroy( & cond);

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

pthread\_mutex\_destroy( & forks[i]);

}

return 0;

}

void \* philosopher(void \* num) {

int id = (int)(long) num;

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

std::cout << "Philosopher " << id << " садится к столу.\n";

pthread\_mutex\_unlock( & cout\_lock);

int right\_fork = id;

int left\_fork = (id + 1) % PHILO;

int first\_fork = std::min(left\_fork, right\_fork);

int second\_fork = std::max(left\_fork, right\_fork);

int f;

while ((f = food\_on\_table()) > 0) {

if (id == 1) {

sleep(sleep\_seconds);

}

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

std::cout << "Philosopher " << id << ": начинает есть " << f << ".\n";

pthread\_mutex\_unlock( & cout\_lock);

get\_forks(id, first\_fork, second\_fork);

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

std::cout << "Philosopher " << id << ": ест.\n";

pthread\_mutex\_unlock( & cout\_lock);

usleep(DELAY \* (FOOD - f + 1));

down\_forks(first\_fork, second\_fork);

}

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

std::cout << "Philosopher " << id << " поел.\n";

pthread\_mutex\_unlock( & cout\_lock);

return nullptr;

}

int food\_on\_table() {

static int food = FOOD;

int myfood;

pthread\_mutex\_lock( & foodlock);

if (food > 0) {

food--;

}

myfood = food;

pthread\_mutex\_unlock( & foodlock);

return myfood;

}

void get\_forks(int phil, int fork1, int fork2) {

bool got\_forks = false;

while (!got\_forks) {

pthread\_mutex\_lock( & forks\_lock); // Захватываем мьютекс для проверки вилок

// Пробуем захватить первую вилку

int first\_locked = pthread\_mutex\_trylock( & forks[fork1]);

if (first\_locked == 0) {

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

std::cout << "Philosopher " << phil << ": взял вилку " << fork1 << ".\n";

pthread\_mutex\_unlock( & cout\_lock);

// Пробуем захватить вторую вилку

int second\_locked = pthread\_mutex\_trylock( & forks[fork2]);

if (second\_locked == 0) {

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

std::cout << "Philosopher " << phil << ": взял вилку " << fork2 << ".\n";

pthread\_mutex\_unlock( & cout\_lock);

got\_forks = true; // Успешно захватили обе вилки

} else {

// Не удалось захватить вторую вилку, освобождаем первую

pthread\_mutex\_unlock( & forks[fork1]);

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

std::cout << "Philosopher " << phil << ": положил вилку " << fork1 << ", ждёт.\n";

pthread\_mutex\_unlock( & cout\_lock);

// Ждём на условной переменной

pthread\_cond\_wait( & cond, & forks\_lock);

}

} else {

// Не удалось захватить первую вилку, ждём

pthread\_cond\_wait( & cond, & forks\_lock);

}

pthread\_mutex\_unlock( & forks\_lock); // Освобождаем мьютекс после проверки

}

}

void down\_forks(int fork1, int fork2) {

pthread\_mutex\_lock( & forks\_lock); // Захватываем мьютекс перед освобождением вилок

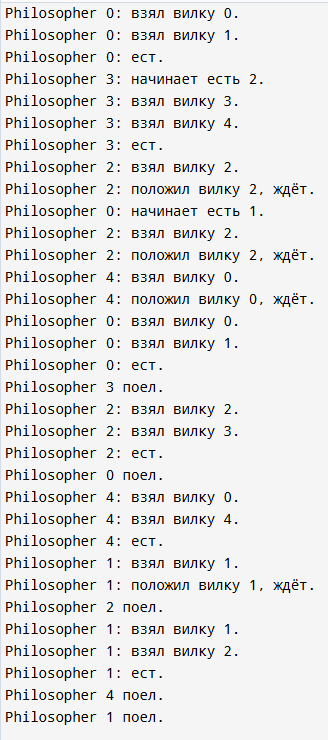
pthread\_mutex\_unlock( & forks[fork1]);

pthread\_mutex\_unlock( & forks[fork2]);

pthread\_cond\_broadcast( & cond); // Оповещаем всех ждущих философов

pthread\_mutex\_unlock( & forks\_lock);

}



**Задание 23**:Sleepsort 2 (огласите весь список)

**Условие**: Переделайте программу упражнения 6 так, чтобы нити не выводили строки в стандартный поток вывода, а формировали из них связный список с порядком записей, соответствующим порядку сортировки. После завершения всех «сортирующих» потоков, одна из потоков (возможно, но не обязательно основная) должна выводить весь список. Обратите внимание, что доступ к списку необходимо синхронизировать.

#include <iostream>

#include <vector>

#include <string>

#include <pthread.h>

#include <unistd.h>

using namespace std;

// Структура для узла связного списка

struct Node {

string str;

Node \* next;

Node(const string & s): str(s), next(nullptr) {}

};

// Структура для передачи данных в поток

struct ThreadData {

string str;

size\_t length;

Node \*\* head; // Указатель на начало списка

Node \*\* tail; // Указатель на конец списка

pthread\_mutex\_t \* list\_mutex; // Мьютекс для списка

int \* finished\_count; // Счётчик завершённых потоков

pthread\_mutex\_t \* count\_mutex; // Мьютекс для счётчика

pthread\_cond\_t \* cond; // Условная переменная для ожидания

int total\_threads; // Общее число потоков

};

// Функция для добавления строки в конец списка

void add\_to\_list(Node \*\* head, Node \*\* tail,

const string & str, pthread\_mutex\_t \* list\_mutex) {

pthread\_mutex\_lock(list\_mutex);

Node \* new\_node = new Node(str);

if ( \* head == nullptr) {

// Список пуст, новый узел становится головой и хвостом

\* head = new\_node;

\* tail = new\_node;

} else {

// Добавляем в конец списка

( \* tail) -> next = new\_node;

\* tail = new\_node;

}

pthread\_mutex\_unlock(list\_mutex);

}

// Функция, выполняемая каждым потоком

void \* sortString(void \* arg) {

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

// Спим время, пропорциональное длине строки (в микросекундах)

usleep(data -> length \* 10000);

// Добавляем строку в конец списка

add\_to\_list(data -> head, data -> tail, data -> str, data -> list\_mutex);

// Увеличиваем счётчик завершённых потоков

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

( \* data -> finished\_count) ++;

if ( \* data -> finished\_count == data -> total\_threads) {

// Если это последний поток, сигнализируем основному потоку

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

}

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

delete data; // Освобождаем память

return nullptr;

}

int main() {

vector < string > strings;

string input;

// Считываем строки до конца ввода

while (getline(cin, input)) {

if (input.empty() && cin.eof()) break;

strings.push\_back(input);

}

const int n = strings.size();

if (n > 100) {

cerr << "Ошибка: допустимо 100 строк максимум" << endl;

return 1;

}

// Инициализация синхронизационных примитивов

pthread\_mutex\_t list\_mutex;

pthread\_mutex\_t count\_mutex;

pthread\_cond\_t cond;

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

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

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

Node \* head = nullptr; // Начало связного списка

Node \* tail = nullptr; // Конец связного списка

int finished\_count = 0; // Счётчик завершённых потоков

// Массив потоков

vector < pthread\_t > threads(n);

// Создаем потоки

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

ThreadData \* data = new ThreadData {

strings[i],

strings[i].length(), &

head, &

tail, &

list\_mutex, &

finished\_count, &

count\_mutex, &

cond,

n

};

int result = pthread\_create( & threads[i], nullptr, sortString, (void \* ) data);

if (result != 0) {

cerr << "Ошибка создания потока " << i << endl;

delete data;

return 1;

}

}

// Ожидаем завершения всех потоков

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

while (finished\_count < n) {

pthread\_cond\_wait( & cond, & count\_mutex);

}

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

// Присоединяем потоки для корректного завершения

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

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

}

// Выводим связный список

Node \* current = head;

while (current != nullptr) {

cout << current -> str << endl;

Node \* temp = current;

current = current -> next;

delete temp; // Освобождаем память

}

// Уничтожаем синхронизационные примитивы

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

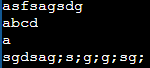
pthread\_mutex\_destroy( & count\_mutex);

pthread\_cond\_destroy( & cond);

return 0;

}

**Input**:



**Output**:

