МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

**Лабораторная работа № 2**

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:** Реализация потоков в C++

**Задание 6:** Алгоритм сортировки sleepsort

**Задание 7:** Многопоточный ср -R

**Задание 8:** Вычислить число Пи

**Задание 9:** Принудительное завершение потока **Задания 8**

**Задание 10:** Обедающие философы

**Ход работы:**

6. Sleepsort

Реализуйте уникальный алгоритм сортировки sleepsort с асимптотикой O(N) (по процессорному времени).

На стандартный вход программы подается не более 100 строк различной длины. Вам необходимо вывести эти строки, отсортированные по длине. Строки одинаковой длины могут выводиться в произвольном порядке.

Для каждой входной строки, создайте поток и передайте ей эту строку в качестве параметра. Поток должна вызвать sleep(2) или usleep(2) с параметром, пропорциональным длине этой строки. Затем поток выводит строку в стандартный поток вывода и завершается. Не следует выбирать коэффициент пропорциональности слишком маленьким, вы рискуете получить некорректную сортировку:

#include <iostream>

#include <vector>

#include <string>

#include <pthread.h>

#include <unistd.h>

using namespace std;

// Структура для передачи данных в поток

struct ThreadData {

string str;

size\_t length;

};

// Функция, выполняемая каждым потоком

void \* printString(void \* arg) {

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

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

usleep(data -> length \* 10000); // Коэффициент 10000 микросекунд

cout << data -> str << endl;

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;

}

// 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()

};

// Создаем поток

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

if (result != 0) {

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

delete data;

return 1;

}

// Отсоединяем поток, чтобы он очищался автоматически после завершения

pthread\_detach(threads[i]);

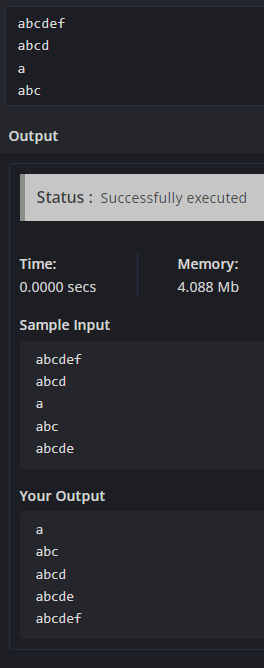
}

// Даем потокам время на завершение

sleep((n + 1) \* 2); // Ждем достаточно долго

return 0;

}



7. Многопоточный ср -R

Реализуйте многопоточную программу рекурсивного копирования дерева подкаталогов, функциональный аналог команды ср(1) с ключом -R. Программа должна принимать два параметра - полное путевое имя корневого каталога исходного дерева и полное путевое имя целевого дерева. Программа должна обходить исходное дерево каталогов при помощи opendir(3c)/readdir\_r(3c) и определять тип каждого найденного файла при помощи stat(2). Для определения размера буфера для readdir\_r используйте pathconf(2) (sizeof(struct dirent) + pathconf(directory)+1).

Для каждого подкаталога должен создаваться одноименный каталог в целевом дереве и запускаться отдельная поток, обходящая этот подкаталог. Для каждого регулярного файла должна запускаться поток, копирующая этот файл в одноименный файл целевого дерева при помощи open(2)/read(2)/write(2). Файлы других типов (символические связи, именованные трубы и др.) следует игнорировать.

При копировании больших деревьев каталогов возможны проблемы с исчерпанием лимита открытых файлов. Очень важно закрывать дескрипторы обработанных файлов и каталогов при помощи сlоѕе(2)/closedir(3c). Тем не менее, для очень больших деревьев этого может оказаться недостаточно. Допускается обход этой проблемы при помощи холостого цикла с ожиданием (если open(2) или readdir(3c) завершается с ошибкой EMFILE, то допускается сделать sleep(3с) и повторить попытку открытия через некоторое время).

Обратите также внимание, что значения дескрипторов открытых файлов могут переиспользоваться, т.е. в разные моменты времени один и тот же дескриптор может указывать на разные файлы. Чтобы избежать связанных с этим проблем, избегайте передачи дескрипторов между нитями. Вся работа с дескриптором от создания до закрытия должна происходить в одной потоки.

Дополнительное упражнение: при помощи команды time(1) сравните ресурсы, потребляемые вашей программой и командой ср -R при копировании одного и того же дерева каталогов. Объясните наблюдаемые различия. Каким образом их можно устранить? Следует ли вообще реализовать копирование файлов таким способом и если да, то в каких условиях?

#include <errno.h>

#include <iostream>

#include <pthread.h>

#include <dirent.h>

#include <sys/stat.h>

#include <unistd.h>

#include <fcntl.h>

#include <string>

#include <cstring>

#include <vector>

struct CopyArgs {

std::string src\_path;

std::string dest\_path;

};

// Функция копирования файла

void \* copy\_file(void \* arg) {

CopyArgs \* args = (CopyArgs \* ) arg;

int src\_fd = open(args -> src\_path.c\_str(), O\_RDONLY);

while (src\_fd == -1 && errno == EMFILE) {

sleep(1); // Ждем при превышении лимита файлов

src\_fd = open(args -> src\_path.c\_str(), O\_RDONLY);

}

if (src\_fd == -1) {

perror("open source file");

delete args;

return nullptr;

}

int dest\_fd = open(args -> dest\_path.c\_str(), O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

while (dest\_fd == -1 && errno == EMFILE) {

sleep(1);

dest\_fd = open(args -> dest\_path.c\_str(), O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

}

if (dest\_fd == -1) {

perror("open destination file");

close(src\_fd);

delete args;

return nullptr;

}

char buffer[4096];

ssize\_t bytes\_read;

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

write(dest\_fd, buffer, bytes\_read);

}

close(src\_fd);

close(dest\_fd);

delete args;

return nullptr;

}

// Функция обхода директории

void \* process\_directory(void \* arg) {

CopyArgs \* args = (CopyArgs \* ) arg;

std::vector < pthread\_t > threads;

DIR \* dir = opendir(args -> src\_path.c\_str());

while (dir == nullptr && errno == EMFILE) {

sleep(1); // Ждем при превышении лимита файлов

dir = opendir(args -> src\_path.c\_str());

}

if (dir == nullptr) {

perror("opendir");

delete args;

return nullptr;

}

long name\_max = pathconf(args -> src\_path.c\_str(), \_PC\_NAME\_MAX);

if (name\_max == -1) name\_max = 255;

size\_t dirent\_size = sizeof(struct dirent) + name\_max + 1;

struct dirent \* entry = (struct dirent \* ) malloc(dirent\_size);

struct dirent \* result;

// Создаем целевую директорию

mkdir(args -> dest\_path.c\_str(), 0755);

while (readdir\_r(dir, entry, & result) == 0 && result != nullptr) {

if (strcmp(entry -> d\_name, ".") == 0 || strcmp(entry -> d\_name, "..") == 0) {

continue;

}

std::string src\_item = args -> src\_path + "/" + entry -> d\_name;

std::string dest\_item = args -> dest\_path + "/" + entry -> d\_name;

struct stat stat\_buf;

if (stat(src\_item.c\_str(), & stat\_buf) == -1) {

continue;

}

if (S\_ISDIR(stat\_buf.st\_mode)) {

// Для директории создаем новый поток

CopyArgs \* dir\_args = new CopyArgs {

src\_item,

dest\_item

};

pthread\_t thread;

pthread\_create( & thread, nullptr, process\_directory, dir\_args);

threads.push\_back(thread);

} else if (S\_ISREG(stat\_buf.st\_mode)) {

// Для обычного файла создаем поток копирования

CopyArgs \* file\_args = new CopyArgs {

src\_item,

dest\_item

};

pthread\_t thread;

pthread\_create( & thread, nullptr, copy\_file, file\_args);

threads.push\_back(thread);

}

// Другие типы файлов игнорируем

}

free(entry);

closedir(dir);

// Ждем завершения всех потоков

for (pthread\_t thread: threads) {

pthread\_join(thread, nullptr);

}

delete args;

return nullptr;

}

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

if (argc != 3) {

std::cerr << "Usage: " << argv[0] << " <source\_dir> <dest\_dir>" << std::endl;

return 1;

}

std::string src\_dir = argv[1];

std::string dest\_dir = argv[2];

CopyArgs \* args = new CopyArgs {

src\_dir,

dest\_dir

};

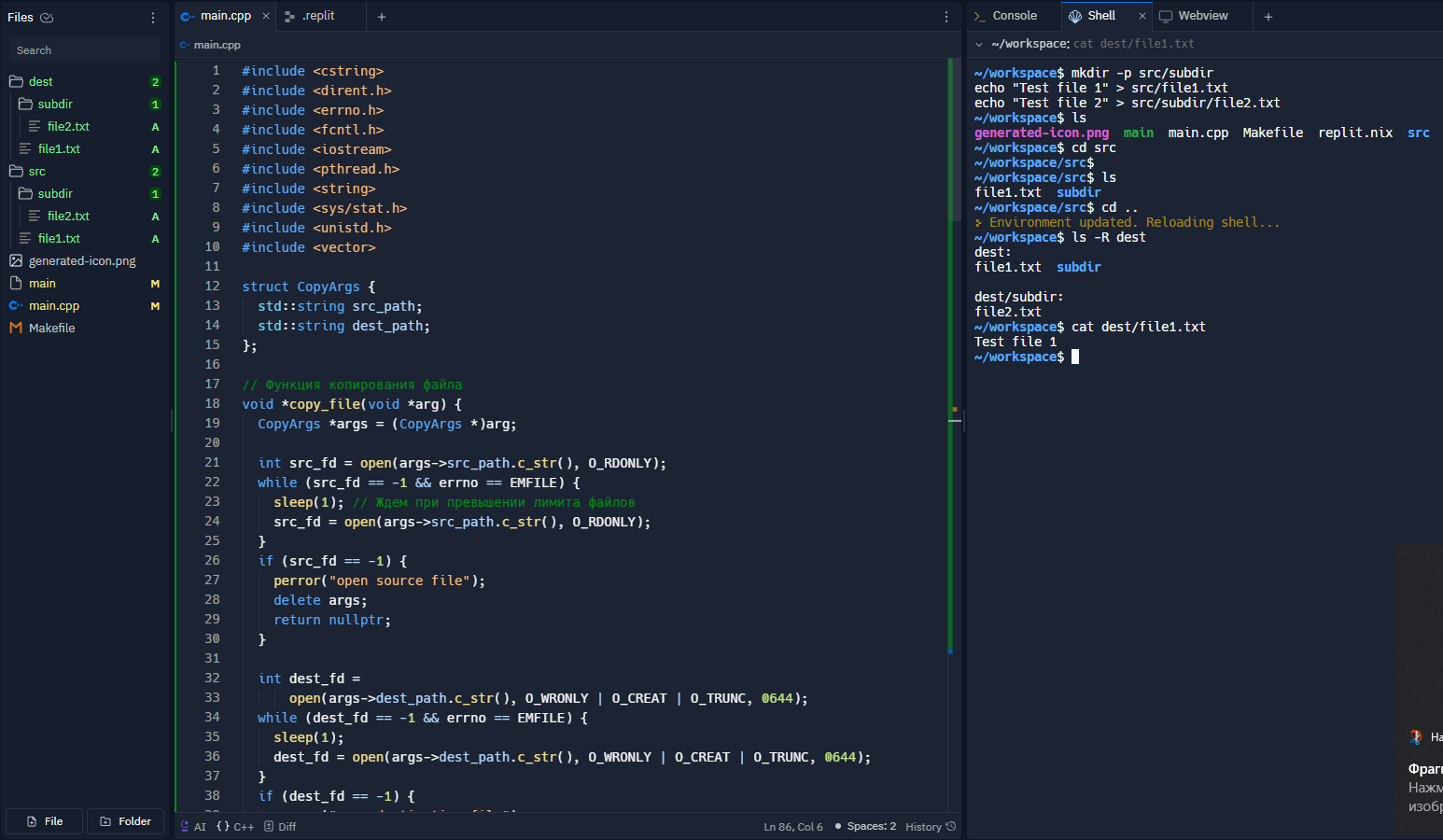
pthread\_t main\_thread;

pthread\_create( & main\_thread, nullptr, process\_directory, args);

pthread\_join(main\_thread, nullptr);

return 0;

}



Я создал сначала репозиторий, а потом скопировал такой же в «dest».

8. Использует ряд Лейбница вычислите число Пи: π/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + ...

Однопоточная версия такой программы доступна в файле pi\_serial.С. Количество потоков программы должно определяться параметром командной строки. Количество итераций может определяться во время компиляции. Для передачи частичных сумм ряда, подсчитанных потоками, используйте pthread\_exit(3c)/pthread\_join(3c). Обратите внимание, что на 32 разрядных платформах sizeof(double)>sizeof(void \*), поэтому частичную сумму ряда нельзя преобразовывать к указателю, для нее надо выделять собственную память:

#include <iomanip>

#include <iostream>

#include <pthread.h>

#include <cstdlib>

#define num\_steps 2000000000

// Структура для передачи аргументов в поток и хранения результата

struct ThreadArg {

long thread\_id;

long num\_threads;

double partial\_sum;

};

// Функция, выполняемая каждым потоком

void \* calculate\_pi(void \* arg) {

ThreadArg \* thread\_arg = (ThreadArg \* ) arg;

long thread\_id = thread\_arg -> thread\_id;

long num\_threads = thread\_arg -> num\_threads;

double partial\_sum = 0.0;

// Каждый поток вычисляет свою часть ряда Лейбница

for (long i = thread\_id; i < num\_steps; i += num\_threads) {

partial\_sum += 1.0 / (i \* 4.0 + 1.0);

partial\_sum -= 1.0 / (i \* 4.0 + 3.0);

}

// Сохраняем частичную сумму в структуре

thread\_arg -> partial\_sum = partial\_sum;

// Завершаем поток, возвращая указатель на структуру

pthread\_exit((void \* ) thread\_arg);

}

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

// Проверяем, передан ли аргумент с количеством потоков

if (argc != 2) {

std::cerr << "Использование: " << argv[0] << " <число\_потоков>\n";

return EXIT\_FAILURE;

}

// Получаем количество потоков из аргумента командной строки

long num\_threads = std::atol(argv[1]);

if (num\_threads <= 0) {

std::cerr << "Число потоков должно быть положительным\n";

return EXIT\_FAILURE;

}

// Создаём массив потоков и аргументов для них

pthread\_t \* threads = new pthread\_t[num\_threads];

ThreadArg \* thread\_args = new ThreadArg[num\_threads];

// Инициализируем аргументы для каждого потока

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

thread\_args[i].thread\_id = i;

thread\_args[i].num\_threads = num\_threads;

thread\_args[i].partial\_sum = 0.0;

// Создаём поток

if (pthread\_create( & threads[i], nullptr, calculate\_pi, (void \* ) & thread\_args[i]) != 0) {

std::cerr << "Ошибка создания потока " << i << "\n";

delete[] threads;

delete[] thread\_args;

return EXIT\_FAILURE;

}

}

// Суммируем результаты всех потоков

double pi = 0.0;

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

void \* thread\_result;

if (pthread\_join(threads[i], & thread\_result) != 0) {

std::cerr << "Ошибка при объединении потока " << i << "\n";

delete[] threads;

delete[] thread\_args;

return EXIT\_FAILURE;

}

// Добавляем частичную сумму к общему результату

ThreadArg \* result = (ThreadArg \* ) thread\_result;

pi += result -> partial\_sum;

}

// Умножаем на 4 для получения числа Пи

pi \*= 4.0;

// Выводим результат

std::cout << "pi done - " << std::fixed << std::setprecision(15) << pi << "\n";

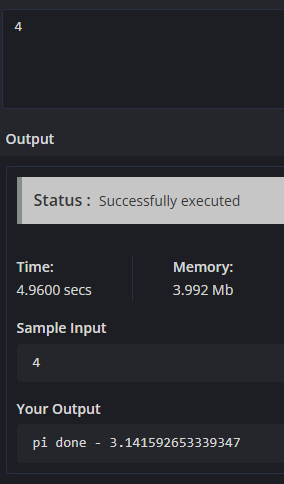
// Освобождаем память

delete[] threads;

delete[] thread\_args;

return EXIT\_SUCCESS;

}



9. Вычисление Пи пока не надоест: Модифицируйте программу упражнения 8 так, чтобы сама по себе она не завершалась. Вместо этого, после нажатия ^-С (после получения сигнала SIGINT) программа должна как можно скорее завершаться, собирать частичные суммы ряда и выводить полученное приближение числа. Рекомендации: ожидаемое решение состоит в том, что вы установите обработчик SIGINT. Этот обработчик должен устанавливать глобальную флаговую переменную. Вычислительные потоки должны просматривать значение флага через некоторое количество итераций, например, через 1000000, и выходить при помощи pthread\_exit, если флаг установлен. Подумайте, как минимизировать ошибку, обусловленную тем, что разные потоки к моменту завершения успели пройти разное количество итераций. Скорее всего, такая минимизация должна обеспечиваться за счет увеличения времени между получением сигнала и выходом:

#include <iomanip>

#include <iostream>

#include <pthread.h>

#include <cstdlib>

#include <csignal>

#include <unistd.h>

// Глобальная переменная-флаг для обработки SIGINT

volatile sig\_atomic\_t stop\_flag = 0;

// Структура для передачи аргументов в поток и хранения результата

struct ThreadArg {

long thread\_id;

long num\_threads;

double partial\_sum;

long iterations\_done; // Количество выполненных итераций

};

// Обработчик сигнала SIGINT

void signal\_handler(int signum) {

stop\_flag = 1;

}

// Функция, выполняемая каждым потоком

void \* calculate\_pi(void \* arg) {

ThreadArg \* thread\_arg = (ThreadArg \* ) arg;

long thread\_id = thread\_arg -> thread\_id;

long num\_threads = thread\_arg -> num\_threads;

double partial\_sum = 0.0;

long i = thread\_id;

long iterations\_done = 0;

// Бесконечный цикл для вычисления ряда Лейбница

while (true) {

// Вычисляем 1,000,000 итераций перед проверкой флага

for (long j = 0; j < 1000000; ++j) {

partial\_sum += 1.0 / (i \* 4.0 + 1.0);

partial\_sum -= 1.0 / (i \* 4.0 + 3.0);

i += num\_threads;

++iterations\_done;

}

// Проверяем флаг остановки

if (stop\_flag) {

break;

}

}

// Сохраняем результаты

thread\_arg -> partial\_sum = partial\_sum;

thread\_arg -> iterations\_done = iterations\_done;

// Завершаем поток

pthread\_exit((void \* ) thread\_arg);

}

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

// Проверяем, передан ли аргумент с количеством потоков

if (argc != 2) {

std::cerr << "Использование: " << argv[0] << " <число\_потоков>\n";

return EXIT\_FAILURE;

}

// Получаем количество потоков из аргумента командной строки

long num\_threads = std::atol(argv[1]);

if (num\_threads <= 0) {

std::cerr << "Число потоков должно быть положительным\n";

return EXIT\_FAILURE;

}

// Устанавливаем обработчик SIGINT

signal(SIGINT, signal\_handler);

// Создаём массив потоков и аргументов для них

pthread\_t \* threads = new pthread\_t[num\_threads];

ThreadArg \* thread\_args = new ThreadArg[num\_threads];

// Инициализируем аргументы для каждого потока

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

thread\_args[i].thread\_id = i;

thread\_args[i].num\_threads = num\_threads;

thread\_args[i].partial\_sum = 0.0;

thread\_args[i].iterations\_done = 0;

// Создаём поток

if (pthread\_create( & threads[i], nullptr, calculate\_pi, (void \* ) & thread\_args[i]) != 0) {

std::cerr << "Ошибка создания потока " << i << "\n";

delete[] threads;

delete[] thread\_args;

return EXIT\_FAILURE;

}

}

// Ожидаем завершения потоков после получения SIGINT

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

void \* thread\_result;

if (pthread\_join(threads[i], & thread\_result) != 0) {

std::cerr << "Ошибка при объединении потока " << i << "\n";

delete[] threads;

delete[] thread\_args;

return EXIT\_FAILURE;

}

// Добавляем частичную сумму к общему результату

ThreadArg \* result = (ThreadArg \* ) thread\_result;

std::cout << "Поток " << i << " выполнил " << result -> iterations\_done << " итераций\n";

thread\_args[i].partial\_sum = result -> partial\_sum;

thread\_args[i].iterations\_done = result -> iterations\_done;

}

// Находим минимальное количество итераций, выполненных всеми потоками

long min\_iterations = thread\_args[0].iterations\_done;

for (long i = 1; i < num\_threads; ++i) {

if (thread\_args[i].iterations\_done < min\_iterations) {

min\_iterations = thread\_args[i].iterations\_done;

}

}

// Корректируем частичные суммы, чтобы учесть только min\_iterations

double pi = 0.0;

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

long excess\_iterations = thread\_args[i].iterations\_done - min\_iterations;

double excess\_sum = 0.0;

// Вычисляем сумму лишних итераций, чтобы вычесть их

long start\_index = i + min\_iterations \* num\_threads;

for (long j = 0; j < excess\_iterations; ++j) {

long idx = start\_index + j \* num\_threads;

excess\_sum += 1.0 / (idx \* 4.0 + 1.0);

excess\_sum -= 1.0 / (idx \* 4.0 + 3.0);

}

// Корректируем частичную сумму

pi += thread\_args[i].partial\_sum - excess\_sum;

}

// Умножаем на 4 для получения числа Пи

pi \*= 4.0;

// Выводим результат

std::cout << "Число итераций (по минимуму): " << min\_iterations \* num\_threads << "\n";

std::cout << "pi done - " << std::fixed << std::setprecision(15) << pi << "\n";

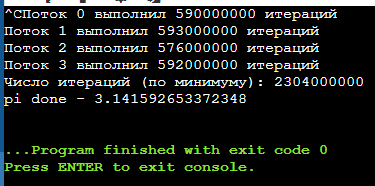
// Освобождаем память

delete[] threads;

delete[] thread\_args;

return EXIT\_SUCCESS;

}



Минимизация ошибки: Чтобы уменьшить ошибку из-за разного количества выполненных итераций, я добавил поле iterations\_done в структуру ThreadArg, которое отслеживает количество итераций, выполненных каждым потоком. После завершения всех потоков программа находит минимальное количество итераций (min\_iterations), выполненных всеми потоками. Затем для каждого потока вычисляется сумма "лишних" итераций (тех, что превышают min\_iterations), и эта сумма вычитается из частичной суммы потока. Таким образом, итоговое значение Пи учитывает только те итерации, которые гарантированно выполнены всеми потоками.

10. Обедающие философы

Возьмите за основу программу din\_phil.С. Эта программа симулирует известную задачу про обедающих философов.

Пять философов сидят за круглым столом и едят спагетти. Спагетти едят при помощи двух вилок. Каждые двое философов, сидящих рядом, пользуются общей вилкой. Философ некоторое время размышляет, потом пытается взять вилки и принимается за еду. Съев некоторое количество спагетти, философ освобождает вилки и снова начинает размышлять. Еще через некоторое время он снова принимается за еду, и т.Д., пока спагетти не кончатся. Если одну из вилок взять не получается, философ ждет, пока она освободится.

В программе din\_phil. С философы симулируются при помощи потоков, периоды размышлений и еды - при помощи usleep(3c), а вилки при помощи мутексов. Философы всегда берут сначала левую вилку, а потом правую. При некоторых обстоятельствах это может приводить к мертвой блокировке. Измените протокол взаимодействия философов с вилками таким образом, чтобы мертвых блокировок не происходило:

#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

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); // Инициализируем мьютекс для вывода

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); // Уничтожаем мьютекс для вывода

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) {

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

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

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

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

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

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

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

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

}

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

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

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

}

**Вывод кода**:

Philosopher 0 садится к столу.

Philosopher 0: начинает есть 49.

Philosopher 0: взял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 4 садится к столу.

Philosopher 2 садится к столу.

Philosopher 2: начинает есть 47.

Philosopher 3 сади�ся к столу.

Philosopher 3: начинает есть 46.

Philosopher 3: взял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 1 садится к столу.

Philosopher 2: взял вилку 2.

Philosopher 4: начинает есть 48.

Philosopher 0: начинает есть 44.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 3: начинает есть 43.

Philosopher 3: в�ял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 0: начинает есть 42.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 3: начинает есть 41.

Philosopher 2: в�ял вилку 3.

Philosopher 2: ест.

Philosopher 2: начинает есть 40.

Philosopher 2: в�ял вилку 2.

Philosopher 3: взял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 0: начинает есть 39.

Philosopher 4: в�ял вилку 0.

Philosopher 3: начинает есть 38.

Philosopher 3: в�ял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 3: начинает есть 37.

Philosopher 2: в�ял вилку 3.

Philosopher 2: ест.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 4: начинает есть 36.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 2: начинает есть 35.

Philosopher 2: в�ял вилку 2.

Philosopher 3: взял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 0: начинает есть 34.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 3: начинает есть 33.

Philosopher 2: в�ял вилку 3.

Philosopher 2: ест.

Philosopher 0: начинает есть 32.

Philosopher 4: в�ял вилку 0.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 2: начинает есть 31.

Philosopher 2: в�ял вилку 2.

Philosopher 3: взял вилку 3.

Philosopher 4: начинает есть 30.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 3: начинает есть 29.

Philosopher 2: в�ял вилку 3.

Philosopher 2: ест.

Philosopher 0: начинает есть 28.

Philosopher 4: взял вилку 0.

Philosopher 4: взял вилку 4.

Philosopher 4: е�т.

Philosopher 2: начинает есть 27.

Philosopher 2: в�ял вилку 2.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 4: начинает есть 26.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 1: начинает есть 45.

Philosopher 2: начинает есть 25.

Philosopher 2: в�ял вилку 2.

Philosopher 3: взял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 0: начинает есть 24.

Philosopher 1: в�ял вилку 1.

Philosopher 4: взял вилку 0.

Philosopher 3: начинает есть 23.

Philosopher 3: в�ял вилку 3.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 3: начинает есть 22.

Philosopher 2: в�ял вилку 3.

Philosopher 2: ест.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 4: начинает есть 21.

Philosopher 4: в�ял вилку 0.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 2: начинает есть 20.

Philosopher 1: в�ял вилку 2.

Philosopher 1: ест.

Philosopher 3: взял вилку 3.

Philosopher 2: взял вилку 2.

Philosopher 4: начинает есть 18.

Philosopher 4: в�ял вилку 0.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 3: начинает есть 17.

Philosopher 4: в�ял вилку 4.

Philosopher 4: ест.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 2: начинает есть 16.

Philosopher 2: в�ял вилку 2.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 4: начинает есть 15.

Philosopher 4: в�ял вилку 0.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 2: начинает есть 14.

Philosopher 2: в�ял вилку 2.

Philosopher 3: взял вилку 3.

Philosopher 4: начинает есть 13.

Philosopher 0: в�ял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 1: начинает есть 19.

Philosopher 0: начинает есть 12.

Philosopher 4: в�ял вилку 0.

Philosopher 1: взял вилку 1.

Philosopher 3: начинает есть 11.

Philosopher 2: в�ял вилку 3.

Philosopher 2: ест.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 2: начинает есть 10.

Philosopher 3: в�ял вилку 3.

Philosopher 1: взял вилку 2.

Philosopher 1: ест.

Philosopher 4: начинает есть 9.

Philosopher 0: взял вилку 0.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 2: взял вилку 2.

Philosopher 3: начинает есть 7.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 0: начинает есть 6.

Philosopher 0: взял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 2: начинает есть 5.

Philosopher 2: взял вилку 2.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 0: начинает есть 4.

Philosopher 0: взял вилку 0.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 2: начинает есть 3.

Philosopher 2: взял вилку 2.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 0: начинает есть 2.

Philosopher 4: взял вилку 0.

Philosopher 4: взял вилку 4.

Philosopher 4: ест.

Philosopher 1: начинает есть 8.

Philosopher 1: взял вилку 1.

Philosopher 2: начинает есть 1.

Philosopher 3: взял вилку 3.

Philosopher 1: взял вилку 2.

Philosopher 1: е�т.

Philosopher 4 поел.

Philosopher 0: взял вилку 0.

Philosopher 3: взял вилку 4.

Philosopher 3: ест.

Philosopher 1 поел.

Philosopher 2: взял вилку 2.

Philosopher 0: взял вилку 1.

Philosopher 0: ест.

Philosopher 3 поел.

Philosopher 2: взял вилку 3.

Philosopher 2: ест.

Philosopher 0 поел.

Philosopher 2 поел.

Такой длинный вывод происходит из-за того, что установлено большое количество еды «const int FOOD = 50;».

Из-за русской кодировки случаются небольшие баги в виде символа �.

Я выбрал подход с упорядочиванием взятия вилок, так как он прост в реализации и эффективно решает проблему deadlock. Каждый философ будет брать вилки в порядке возрастания их номеров: сначала вилку с меньшим индексом, затем с большим. Это гарантирует, что не возникнет циклического ожидания.