|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра прикладной математики | | | |
|  | | | |
| Лабораторная работа № 2 | | | |
| по дисциплине «Параллельное программирование» | | | |
| **ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВУЮЩИХ ПОТОКОВ** | | | |
|  | | | |
|  | Бригада 6 | ПМИ-71 Антонов С. |
|  | ПМИ-71 Арнольд э. |
|  | ПМИ-71 Кайль Д. |
|  |
|  |
| Преподаватель | Щукин Г. А.  Городничев М. А. |
|  | | | |
| Новосибирск | | | |

**Задание**

1. Изучить теоретическую часть из раздела 2 и 3 настоящего пособия.

2. Записать программу из примера 4. Скомпилировать ее и проверить корректность работы.

3. Добавить в пример 4 вывод информации о том, какой поток какую задачу взял на исполнение, а также некоторое достаточно продолжительное вычисление в функцию do\_task(). Проверить, что все задачи выполняются, причем каждая – не более одного раза. Отключить вызовы функций pthread\_mutex\_lock() и pthread\_mutex\_unlock() и убедиться, что при отсутствии мьютекса некоторые задачи будут выполняться дважды, тогда как другие не выполнятся ни разу.

4. Сравнить скорости работы примитивов синхронизации – мьютекса и спинлока. Выяснить, как количество потоков, одновременно обращающихся к ресурсу, защищенному примитивом синхронизации,влияет на скорость исполнения программы.

5. Реализовать условную переменную с помощью мьютекса и цикла ожидания.

6. Реализовать функцию для выполнения вычислений по модели MapReduce. Подробную информацию о модели MapReduce можнонайти в электронном документе: Калита Р. MapReduce5. В качествепараметров функция должна принимать массив данных для обработки,имена функций map и reduce, количество разрешенных потоков.В качестве реализации параллельного применения функции map к каж-дому элементу массива использовать результаты пункта 8 порядка выполнения лабораторной работы № 1. Реализовать параллельное применение функции reduce к списку, полученному в результате работы функции map.

7. Выполнить задание для самостоятельной работы (вариант согласовывается с преподавателем), по результатам подготовить отчет.

**Задание для самостоятельной работы**

Пул потоков – это набор потоков, которые могут использоваться для решения различных задач.

Задачи помещаются в общую очередь. Свободный поток (поток, который не решает в настоящий момент никакую задачу) обращается к очереди для получения очередной задачи, выполняет ее, после чего запрашивает в очереди следующую задачу. В результате исполнения некоторой задачи могут появляться новые задачи. Они также добавляются в очередь. Программа завершается, когда в очереди не остается задач и все потоки оказываются свободными.

Необходимо реализовать пул потоков и на его основе решить задачу: генерации простых чисел с помощью алгоритма «Решето Эратосфена».

Из вышеперечисленных 1-7 заданий можно сформировать 4:

1. Добавить в пример 4 вывод информации о том, какой поток какую задачу взял на исполнение, а также некоторое достаточно продолжительное вычисление в функцию do\_task(). Проверить, что все задачи выполняются, причем каждая – не более одного раза. Отключить вызовы функций pthread\_mutex\_lock() и pthread\_mutex\_unlock() и убедиться, что при отсутствии мьютекса некоторые задачи будут выполняться дважды, тогда как другие не выполнятся ни разу.

2. Сравнить скорости работы примитивов синхронизации – мьютекса и спинлока. Выяснить, как количество потоков, одновременно обращающихся к ресурсу, защищенному примитивом синхронизации, влияет на скорость исполнения программы.

3. Реализовать условную переменную с помощью мьютекса и цикла ожидания.

4. Реализовать функцию для выполнения вычислений по модели MapReduce. Подробную информацию можно найти в электронном документе: Калита Р. MapReduce. В качестве параметров функция должна принимать массив данных для обработки, имена функций map и reduce, количество разрешенных потоков. В качестве реализации параллельного применения функции map к каждому элементу массива использовать результаты пункта 2 порядка выполнения лабораторной работы № 1. Реализовать параллельное применение функции reduce к списку, полученному в результате работы функции map.

**Программа**

**cmpsyncprimitives**

*cmpsyncprimitives.c*

#include "cmpsyncprimitives.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <time.h>

#define BILLION 1.0E+9

// Функция обработки ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

// Функция вычета разности между временными величинами

#define clocktimeDifference(start, stop)            \

    1.0 \* (stop.tv\_sec - start.tv\_sec) +            \

    1.0 \* (stop.tv\_nsec - start.tv\_nsec) / BILLION

// Указатель на функция для работы с примитивом

typedef int (\*primitive\_func)();

// Структура для работы с примитивом

typedef struct PrimitiveType {

    primitive\_func init\_func;       // Ф-ия инициализации

    void\* init\_arg;                 // Аргументы инициализации

    primitive\_func destroy\_func;    // Ф-ия освобождения ресурсов

    primitive\_func lock\_func;       // Ф-ия блокировки

    primitive\_func unlock\_func;     // Ф-ия разблокировки

    void\* type;                     // Примитив

} PrimitiveType;

// Аргументы потока

typedef struct ThreadArg {

    primitive\_func lock\_func;       // Ф-ия блокировки

    primitive\_func unlock\_func;     // Ф-ия разблокировки

    void\* type;                     // Примитив

    size\_t\* shared\_data;            // Общие данные

} ThreadArg;

// Функция потока, доступа к примитиву

static void\* threadFunc(void\* arg) {

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

    int err = 0;

    // Получить доступ к примитиву LOCKS\_COUNT раз

    for (size\_t i = 0; i != LOCKS\_COUNT; ++i) {

        err = thread\_arg->lock\_func(thread\_arg->type);

        if (err != 0)

            throwErr("Error: cannot lock primitive!");

        ++\*thread\_arg->shared\_data;

        err = thread\_arg->unlock\_func(thread\_arg->type);

        if (err != 0)

            throwErr("Error: cannot unlock primitive");

    }

    pthread\_exit(NULL);

}

// Инициализация структуры для работы с примитивом

static void primitiveTypeInit(PrimitiveType\* primitive\_type, SyncPrimitive primitive) {

    switch (primitive) {

        case MUTEX: {

            primitive\_type->init\_func    = pthread\_mutex\_init;

            primitive\_type->init\_arg     = NULL;

            primitive\_type->destroy\_func = pthread\_mutex\_destroy;

            primitive\_type->lock\_func    = pthread\_mutex\_lock;

            primitive\_type->unlock\_func  = pthread\_mutex\_unlock;

            primitive\_type->type         = malloc(sizeof(pthread\_mutex\_t));

            break;

        }

        case SPINLOCK: {

            primitive\_type->init\_func    = pthread\_spin\_init;

            primitive\_type->init\_arg     = (void\*)PTHREAD\_PROCESS\_PRIVATE;

            primitive\_type->destroy\_func = pthread\_spin\_destroy;

            primitive\_type->lock\_func    = pthread\_spin\_lock;

            primitive\_type->unlock\_func  = pthread\_spin\_unlock;

            primitive\_type->type         = malloc(sizeof(pthread\_spinlock\_t));

            break;

        }

    }

    if (primitive\_type->type == NULL)

        throwErr("Error: primitive type out of memmory!");

}

// Время работы примитива с указанной функцией, кол-вом потоков

double primitiveTimeStat(SyncPrimitive primitive, uint8\_t threads\_count) {

    PrimitiveType primitive\_type;

    int err = 0;

    size\_t shared\_data = 0;

    // Инициализация примитива

    primitiveTypeInit(&primitive\_type, primitive);

    err = primitive\_type.init\_func(primitive\_type.type, primitive\_type.init\_arg);

    if (err != 0)

        throwErr("Error: cannot initialize primitive");

    // Формирование потоков и их аргументов

    pthread\_t\* threads = (pthread\_t\*)malloc(sizeof(pthread\_t) \* threads\_count);

    if (threads == NULL)

        throwErr("Error: threads out of memmory!");

    ThreadArg threads\_arg = (ThreadArg){primitive\_type.lock\_func,

                                        primitive\_type.unlock\_func,

                                        primitive\_type.type, &shared\_data};

    struct timespec start, stop;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    for (uint8\_t i = 0; i != threads\_count; ++i) {

        err = pthread\_create(&threads[i], NULL, threadFunc, (void\*)&threads\_arg);

        if (err != 0)

            throwErr("Error: cannot create a thread");

    }

    for (uint8\_t i = 0; i != threads\_count; ++i) {

        err = pthread\_join(threads[i], NULL);

        if (err != 0)

            throwErr("Error: cannot join a thread");

    }

    clock\_gettime(CLOCK\_MONOTONIC, &stop);

    double elapsed\_time = clocktimeDifference(start, stop);

    primitive\_type.destroy\_func(primitive\_type.type);

    free(primitive\_type.type);

    free(threads);

    return elapsed\_time;

}

*cmpsyncprimitives.h*

#ifndef CMPSYNCPRIMITIVES\_H

#define CMPSYNCPRIMITIVES\_H

#include <stdint.h>

// Количество блокировок потока (доступов к общим данным)

#define LOCKS\_COUNT 100000000

// Тип используемого примитива

typedef enum SyncPrimitive {

    MUTEX,

    SPINLOCK

} SyncPrimitive;

// Время работы примитива с указанной функцией, кол-вом потоков

double primitiveTimeStat(SyncPrimitive primitive, uint8\_t threads\_count);

#endif

*cmpsyncprimitives\_main.c*

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <float.h>

#include "cmpsyncprimitives.h"

#define ARGS\_COUNT 4

#define TABLE\_FILENAME "result.txt"

// Тип результата тестирования

typedef enum ResultType {

    RT\_AVG,                 // Среднее время

    RT\_MIN                  // Минимальное время

} ResultType;

// Вывод результатов в текстовый файл

static void tableOutput(FILE\* fp, ResultType result\_type, size\_t threads\_count, size\_t measure\_count) {

    double time\_init = 0;

    switch (result\_type) {

        case RT\_AVG: time\_init = 0;       break;

        case RT\_MIN: time\_init = DBL\_MAX; break;

        default: result\_type = RT\_AVG;

    }

    fprintf(fp, "thrds:\tmutex:\tspin:\n");

    for (size\_t i = 0; i != threads\_count; ++i) {

        double mutex\_elapsed\_time = time\_init;

        double spin\_elapsed\_time = time\_init;

        for (size\_t j = 0; j != measure\_count; ++j) {

            double mutex\_tmp = primitiveTimeStat(MUTEX, i + 1);

            double spin\_tmp = primitiveTimeStat(SPINLOCK, i + 1);

            switch (result\_type) {

                case RT\_AVG: {

                    mutex\_elapsed\_time += mutex\_tmp;

                    spin\_elapsed\_time += spin\_tmp;

                    break;

                }

                case RT\_MIN: {

                    if (mutex\_tmp < mutex\_elapsed\_time)

                        mutex\_elapsed\_time = mutex\_tmp;

                    if (spin\_tmp < spin\_elapsed\_time)

                        spin\_elapsed\_time = spin\_tmp;

                    break;

                }

            }

        }

        if (result\_type == RT\_AVG) {

            mutex\_elapsed\_time /= measure\_count;

            spin\_elapsed\_time /= measure\_count;

        }

        fprintf(fp, "%zu\t%lf\t%lf\n", i + 1, mutex\_elapsed\_time, spin\_elapsed\_time);

    }

}

int main(int argc, char\* argv[]) {

    if (argc < ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <threads count> <measure count> <result type>\n");

        fprintf(stderr, "(Result type: 0 - avg; 1 - min)\n");

        exit(EXIT\_FAILURE);

    }

    uint8\_t threads\_count = atoi(argv[1]);

    size\_t measure\_count = atol(argv[2]);

    uint8\_t result\_type = atoi(argv[3]);

    FILE\* fp = fopen(TABLE\_FILENAME, "w");

    tableOutput(fp, result\_type, threads\_count, measure\_count);

    fclose(fp);

    return 0;

}

*Makefile*

DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE

CFLAGS  = -std=c99 -O2 -g $(DEFINES)

LIBS    = -lpthread -lm

TARGET  = cmpsyncprimitives

all: $(TARGET) cleanTemp

$(TARGET): $(TARGET)\_main.o $(TARGET).o

    gcc $(CFLAGS) -o $(TARGET) $(TARGET)\_main.o $(TARGET).o $(LIBS)

$(TARGET)\_main.o: $(TARGET)\_main.c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET)\_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET).c

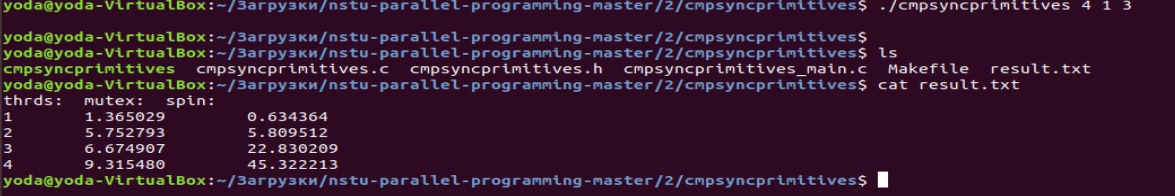
cleanTemp:

    rm -rf \*.o

clean:

    rm -rf $(TARGET)

Результат работы первой задачи



**Condvarimitation**

*condvarimitation.c*

#include "condvarimitation.h"

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <unistd.h>

// Функция обработки ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

// Узел очереди блокировок

// (Состоянние заблокировванного потока)

typedef struct CondNode {

    bool\* state;

    struct CondNode\* next;

} CondNode;

// Очередь блокировок

struct CondQueue {

    struct CondNode\* head;

    struct CondNode\* tail;

};

// Добавление состояние поток в очередь блокировок

static void condQueueSet(CondQueue\* queue, bool\* state) {

    CondNode\* new\_node = (CondNode\*)malloc(sizeof(CondNode));

    if (new\_node == NULL)

        throwErr("Error: new node out of memmory!");

    new\_node->state = state;

    new\_node->next = NULL;

    if (queue->head) {

        queue->tail->next = new\_node;

        queue->tail = new\_node;

    }

    else

        queue->head = queue->tail = new\_node;

}

// Взятие состояние поток из очереди блокировок

static bool\* condQueueGet(CondQueue\* queue) {

    bool\* state = NULL;

    if (queue->head) {

        state = queue->head->state;

        CondNode\* delete\_node = queue->head;

        if (queue->head != queue->tail)

            queue->head = queue->head->next;

        else

            queue->head = queue->tail = NULL;

        free(delete\_node);

    }

    return state;

}

// Инициализация условной переменнной

void condVarInit(CondVar\* cond) {

    int err = pthread\_mutex\_init(&cond->mutex, NULL);

    if (err != 0)

        throwErr("Error: cannot initialize mutex!");

    cond->queue = (CondQueue\*)malloc(sizeof(CondQueue));

    if (cond->queue == NULL)

        throwErr("Error: condition queue out of memmory!");

    cond->queue = (CondQueue\*)malloc(sizeof(CondQueue));

}

// Блокировка до наступления события

void condVarWait(CondVar\* cond, pthread\_mutex\_t\* mutex) {

    // Блокируем условную переменную

    int err = pthread\_mutex\_lock(&cond->mutex);

    if (err != 0)

        throwErr("Error: cannot lock condition mutex!");

    // Сохраняем состояние потока

    bool\* lock = (bool\*)malloc(sizeof(bool));

    \*lock = true;

    condQueueSet(cond->queue, lock);

    // Освобождаем условную переменную

    err = pthread\_mutex\_unlock(&cond->mutex);

    if (err != 0)

        throwErr("Error: cannot unlock condition mutex!");

    // Освобождаем мьютекс до получения сигнала

    err = pthread\_mutex\_unlock(mutex);

    if (err != 0)

        throwErr("Error: cannot unlock mutex!");

    // Ждём пока не получен сигнал

    while (\*lock)

        usleep(WAIT\_TIME\_MS);

    free(lock);

    // Блокируем мьютекс после получения сигнала

    err = pthread\_mutex\_lock(mutex);

    if (err != 0)

        throwErr("Error: cannot lock mutex!");

}

// Сигнал для выхода из блокировки одного потока

void condVarSignal(CondVar\* cond) {

    // Блокируем условную переменную

    int err = pthread\_mutex\_lock(&cond->mutex);

    if (err != 0)

        throwErr("Error: cannot lock condition mutex!");

    // Посылаем сигнал хотя бы одному потоку

    bool\* lock = condQueueGet(cond->queue);

    if (lock)

        \*lock = false;

    // Освобождаем условную переменную

    err = pthread\_mutex\_unlock(&cond->mutex);

    if (err != 0)

        throwErr("Error: cannot unlock condition mutex!");

}

// Сигнал для выхода из блокировки всех потоков

void condVarBroadcast(CondVar\* cond) {

    // Блокируем условную переменную

    int err = pthread\_mutex\_lock(&cond->mutex);

    if (err != 0)

        throwErr("Error: cannot lock condition mutex!");

    // Посылаем сигнал всем потокам

    bool\* lock = NULL;

    while ((lock = condQueueGet(cond->queue)))

        \*lock = false;

    // Освобождаем условную переменную

    err = pthread\_mutex\_unlock(&cond->mutex);

    if (err != 0)

        throwErr("Error: cannot unlock condition mutex!");

}

// Уничтожение условной переменнной

void condVarDestroy(CondVar\* cond) {

    pthread\_mutex\_destroy(&cond->mutex);

    while (condQueueGet(cond->queue));

    free(cond->queue);

}

*condvarimitation.h*

#ifndef CONDVARIMITATION\_H

#define CONDVARIMITATION\_H

#include <pthread.h>

// Время ожидания, между проверками сигнала

// для разблокировки переменной

#define WAIT\_TIME\_MS 1E+6

// Очередь блокировок

typedef struct CondQueue CondQueue;

// Тип условной переменной

typedef struct CondVar {

    pthread\_mutex\_t mutex;

    struct CondQueue\* queue;

} CondVar;

// Инициализация условной переменнной

void condVarInit(CondVar\* cond);

// Блокировка до наступления события

void condVarWait(CondVar\* cond, pthread\_mutex\_t\* mutex);

// Сигнал для выхода из блокировки хотя бы одного потока

void condVarSignal(CondVar\* cond);

// Сигнал для выхода из блокировки всех потоков

void condVarBroadcast(CondVar\* cond);

// Уничтожение условной переменнной

void condVarDestroy(CondVar\* cond);

#endif

*example7.c*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include "condvarimitation.h"

#define err\_exit(code, str) do {                        \

    fprintf(stderr, "%s: %s\n", str, strerror(code));   \

    exit(EXIT\_FAILURE);                                 \

} while (0);

int store;

enum store\_state {EMPTY, FULL} state = EMPTY;

pthread\_mutex\_t mutex;

#ifdef MYCONDVAR

CondVar cond;

#else

pthread\_cond\_t cond;

#endif

void\* producer(void \*arg) {

    int err;

    while (1) {

        // Захватываем мьютекс и ожидаем освобождения склада

        err = pthread\_mutex\_lock(&mutex);

        if (err != 0)

            err\_exit(err, "Cannot lock mutex");

        while (state == FULL) {

            #ifdef MYCONDVAR

            condVarWait(&cond, &mutex);

            #else

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

            if (err != 0)

                err\_exit(err, "Cannot wait on condition variable");

            #endif

        }

        // Получен сигнал, что на складе не осталось товаров.

        // Производим новый товар.

        store = rand();

        state = FULL;

        printf("producing number %d\n", store);

        // Посылаем сигнал, что на складе появился товар.

        #ifdef MYCONDVAR

        condVarSignal(&cond);

        #else

        err = pthread\_cond\_signal(&cond);

        if (err != 0)

            err\_exit(err, "Cannot send signal");

        #endif

        err = pthread\_mutex\_unlock(&mutex);

        if (err != 0)

            err\_exit(err, "Cannot unlock mutex");

    }

}

void\* consumer(void\* arg) {

    int err;

    while (1) {

        // Захватываем мьютекс и ожидаем появления товаров на складе

        err = pthread\_mutex\_lock(&mutex);

        if (err != 0)

            err\_exit(err, "Cannot lock mutex");

        while (state == EMPTY) {

            #ifdef MYCONDVAR

            condVarWait(&cond, &mutex);

            #else

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

            if (err != 0)

                err\_exit(err, "Cannot wait on condition variable");

            #endif

        }

        // Получен сигнал, что на складе имеется товар.

        // Потребляем его.

        printf("Consuming number %d ", store);

        sleep(1);

        printf("done\n");

        state = EMPTY;

        // Посылаем сигнал, что на складе не осталось товаров.

        #ifdef MYCONDVAR

        condVarSignal(&cond);

        #else

        err = pthread\_cond\_signal(&cond);

        if (err != 0)

            err\_exit(err, "Cannot send signal");

        #endif

        err = pthread\_mutex\_unlock(&mutex);

        if (err != 0)

            err\_exit(err, "Cannot unlock mutex");

    }

}

int main(int argc, char\* argv[]) {

    pthread\_t thread1, thread2;

    int err;

    #ifdef MYCONDVAR

    condVarInit(&cond);

    #else

    err = pthread\_cond\_init(&cond, NULL);

    if (err != 0)

        err\_exit(err, "Cannot initialize condition variable");

    #endif

    err = pthread\_mutex\_init(&mutex, NULL);

    if (err != 0)

        err\_exit(err, "Cannot initialize mutex");

    // Создаём потоки

    err = pthread\_create(&thread1, NULL, producer, NULL);

    if(err != 0)

        err\_exit(err, "Cannot create thread 1");

    err = pthread\_create(&thread2, NULL, consumer, NULL);

    if(err != 0)

        err\_exit(err, "Cannot create thread 2");

    // Дожидаемся завершения потоков

    pthread\_join(thread1, NULL);

    pthread\_join(thread2, NULL);

    // Освобождаем ресурсы, связанные с мьютексом

    // и условной переменной

    pthread\_mutex\_destroy(&mutex);

    #ifdef MYCONDVAR

    condVarDestroy(&cond);

    #else

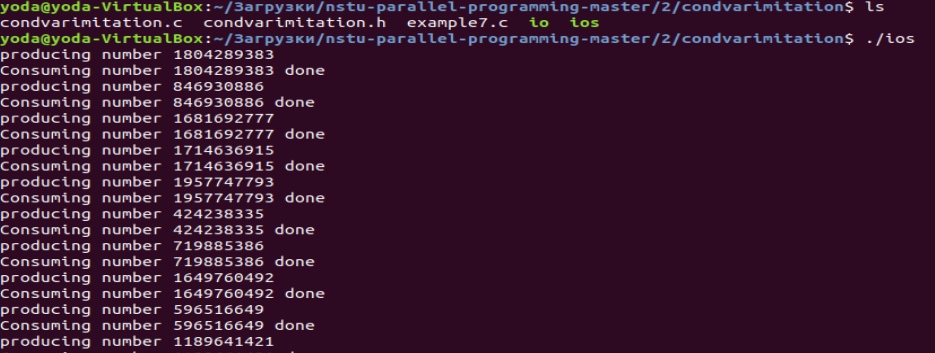
    pthread\_cond\_destroy(&cond);

    #endif

    return 0;

}

Результаты работы второй задачи



**Mapreduce**

*mapreduce.c*

#include "mapreduce.h"

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include <pthread.h>

// Функция обработки ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

// Указатель на фукнцию map/reduce

typedef void (\*func\_ptr)();

// Аргументы потока

typedef struct ThreadArg {

    func\_ptr func;              // Указатель на функцию map/reduce

    void\* arg;                  // Аргумент функции

} ThreadArg;

// Данные программы

typedef struct Data {

    void\* val;                  // Данные

    size\_t val\_size;            // Размер типа данных

    size\_t size;                // Размер данных

    uint8\_t threads\_count;      // Кол-во потоков

    pthread\_t\* threads;

    ThreadArg\* threads\_arg;

    MapArg\* map\_data;           // Аргументы функции map

    ReduceArg\* reduce\_data;     // Аргументы функции reduce

    KeyValNode\*\* collections;   // Массив ключей значений

} Data;                         // (Результаты всех потоков после выполнения функции reduce)

// Инициализация данных

static void dataInit(Data\* data, void\* val, size\_t val\_size, size\_t size, uint8\_t threads\_count) {

    data->val = val;

    data->val\_size = val\_size;

    data->size = size;

    data->threads\_count = threads\_count < size ? threads\_count : size;

}

// Инициализация потоков

static void threadsInit(Data\* data) {

    data->threads = (pthread\_t\*)malloc(sizeof(pthread\_t) \* data->threads\_count);

    if (data->threads == NULL)

        throwErr("Error: threads out of memmory!");

    data->threads\_arg = (ThreadArg\*)malloc(sizeof(ThreadArg) \* data->threads\_count);

    if (data->threads\_arg == NULL)

        throwErr("Error: threads arg out of memmory!");

}

// Инициализация данных функции map

static void mapDataInit(Data\* data) {

    data->map\_data = (MapArg\*)malloc(sizeof(MapArg) \* data->threads\_count);

    if (data->map\_data == NULL)

        throwErr("Error: mad data out of memmory!");

    // Определение размера блоков данных

    size\_t chunk\_size = data->size / data->threads\_count;

    uint8\_t remainder = data->size % data->threads\_count;

    size\_t shift = 0;

    for (uint8\_t i = 0; i != data->threads\_count; ++i) {

        data->map\_data[i].val = data->val + data->val\_size \* shift;

        data->map\_data[i].size = chunk\_size + (i < remainder ? 1 : 0);

        data->map\_data[i].key\_val = NULL;

        shift += data->map\_data[i].size;

    }

}

// Инициализация данных функции reduce

static void reduceDataInit(Data\* data) {

    data->reduce\_data = (ReduceArg\*)malloc(sizeof(ReduceArg) \* data->threads\_count);

    if (data->reduce\_data == NULL)

        throwErr("Error: reduce out of memmory!");

    // Передача списка ключей-значений из функции map в функцию reduce

    for (size\_t i = 0; i != data->threads\_count; ++i) {

        data->reduce\_data[i].key\_val = data->map\_data[i].key\_val;

        data->reduce\_data[i].collection = NULL;

    }

}

static void\* threadFunc(void\* arg) {

    ThreadArg\* chunk = (ThreadArg\*)arg;

    chunk->func(chunk->arg);

    pthread\_exit(NULL);

}

// Выполнение функции map/reduce

static void releaseFunc(Data\* data, func\_ptr func, void\* arg, size\_t arg\_size) {

    int err = 0;

    for (uint8\_t i = 0; i != data->threads\_count; ++i) {

        // Инициализация потоков переданными данными

        data->threads\_arg[i].func = func;                       // Функция

        data->threads\_arg[i].arg = arg + arg\_size \* i;          // Аргумент функции

        err = pthread\_create(&data->threads[i], NULL, threadFunc, (void\*)&data->threads\_arg[i]);

        if (err != 0)

            throwErr("Error: cannot create a map thread!");

    }

    for (uint8\_t i = 0; i != data->threads\_count; ++i) {

        err = pthread\_join(data->threads[i], NULL);

        if (err != 0)

            throwErr("Error: cannot join a map thread!");

    }

}

// Формирование коллекциии результатов после выполнения функции reduce

static void makeCollections(Data\* data) {

    // Создание массива результатотв потоков

    data->collections = (KeyValNode\*\*)malloc(sizeof(KeyValNode\*) \* data->threads\_count);

    if (data->collections == NULL)

        throwErr("Error: collections out of memmory!");

    for (uint8\_t i = 0; i != data->threads\_count; ++i)

        data->collections[i] = data->reduce\_data[i].collection;

}

// Очистка списков ключей-значений

static void keyValClear(KeyValNode\* key\_val\_data, bool clear\_val) {

    KeyValNode\* iter = key\_val\_data;

    while (iter) {

        KeyValNode\* prev = iter;

        iter = iter->next;

        if (prev->key) {

            free(prev->key);

            prev->key = NULL;

        }

        if (clear\_val)

            if (prev->val) {

                free(prev->val);

                prev->key = NULL;

            }

        free(prev);

    }

}

// Очистка всех данных

static void dataClear(Data\* data) {

    free(data->threads);

    free(data->threads\_arg);

    for (uint8\_t i = 0; i != data->threads\_count; ++i) {

        keyValClear(data->map\_data[i].key\_val, false);

        keyValClear(data->reduce\_data[i].collection, true);

    }

    free(data->map\_data);

    free(data->reduce\_data);

    free(data->collections);

}

// Запись результата функций map/reduce

void emitIntermediate(KeyValNode\*\* result, void\* key, void\* val, size\_t size) {

    KeyValNode\* new\_node = (KeyValNode\*)malloc(sizeof(KeyValNode));

    if (new\_node == NULL)

        throwErr("Error: new node out of memmory!");

    new\_node->key = key;

    new\_node->val = val;

    new\_node->size = size;

    new\_node->next = \*result;

    \*result = new\_node;

}

// Обработка массива по модели mapReduce

void\* mapReduceChunk(void\* val, size\_t val\_size, size\_t size,

                     map\_ptr map, reduce\_ptr reduce, merge\_ptr merge,

                     uint8\_t threads\_count) {

    Data data;

    dataInit(&data, val, val\_size, size, threads\_count);

    threadsInit(&data);

    mapDataInit(&data);

    releaseFunc(&data, map, data.map\_data, sizeof(MapArg));

    reduceDataInit(&data);

    releaseFunc(&data, reduce, data.reduce\_data, sizeof(ReduceArg));

    makeCollections(&data);

    void\* result = merge(data.collections, data.threads\_count);

    dataClear(&data);

    return result;

}

*mapreduce.h*

#ifndef MAPREDUCE\_H

#define MAPREDUCE\_H

#include <stdlib.h>

#include <stdint.h>

// Результать работы функций map/reduce

typedef struct KeyValNode {

    void\* key;                  // Ключ

    void\* val;                  // Данные

    size\_t size;                // Размер

    struct KeyValNode\* next;    // Следующий элемент

} KeyValNode;

// Аргумент функций map

typedef struct MapArg {

    // Входные данные:

    void\* val;                  // Данные

    size\_t size;                // Размер

    // Выходные данные:

    KeyValNode\* key\_val;        // Список ключей-значений

} MapArg;

// Аргумент функций reduce

typedef struct ReduceArg {

    // Входные данные:

    KeyValNode\* key\_val;        // Список ключей-значений из функции map

    // Выходные данные:

    KeyValNode\* collection;     // Сжатый список ключей значений

} ReduceArg;

// Указатели на функцию map, reduce, merge

typedef void (\*map\_ptr)(MapArg\*);

typedef void (\*reduce\_ptr)(ReduceArg\*);

typedef void\* (\*merge\_ptr)(KeyValNode\*\*, uint8\_t);

// Запись результата функций map/reduce

void emitIntermediate(KeyValNode\*\* result, void\* key, void\* val, size\_t size);

// Обработка массива по модели mapReduce

void\* mapReduceChunk(void\* val, size\_t val\_size, size\_t size,

                     map\_ptr map, reduce\_ptr reduce, merge\_ptr merge,

                     uint8\_t threads\_count);

#endif

*mapreduce\_main.c*

#include <stdio.h>

#include <time.h>

#include "mapreduce.h"

#define DEMO\_ARGS\_COUNT 3

#define TEST\_ARGS\_COUNT 5

#define RESULT\_FILENAME "result.txt"

#define BILLION 1.0E+9

#define clocktimeDifference(start, stop)            \

    1.0 \* (stop.tv\_sec - start.tv\_sec) +            \

    1.0 \* (stop.tv\_nsec - start.tv\_nsec) / BILLION

// Функция map

// Входные данные:

//     void\* data;               Массив данных

//     size\_t size;              размер массива

// Выходные данные:

//     KeyVal\* key\_val;          Список ключей-значений

void map(MapArg\* arg) {

    // Преобразование массива к исходному типу

    double\* val = (double\*)arg->val;

    for (size\_t i = 0; i != arg->size; ++i)

        ++val[i];

    uint8\_t\* key = (uint8\_t\*)malloc(sizeof(uint8\_t));

    \*key = 1;

    // Добавление результата к списку ключей-значений

    emitIntermediate(&arg->key\_val, (void\*)key, (void\*)val, arg->size);

}

// Функция reduce

// Входные данные:

//     KeyVal\* key\_val;          Список ключей-значений

// Выходные данные:

//     KeyVal\* collection;       Результат ключей-значений

void reduce(ReduceArg\* arg) {

    // Создание результат типа double\*

    double\* result = (double\*)malloc(sizeof(double));

    \*result = 1;

    // Прохождение по всему списку ключей-значений

    KeyValNode\* iter = arg->key\_val;

    while (iter) {

        uint8\_t key = \*(uint8\_t\*)iter->key;

        // Проверка ключа для демонстрации

        if (key == 1) {

            double\* iter\_val = (double\*)iter->val;

            for (size\_t i = 0; i != iter->size; ++i)

                \*result \*= iter\_val[i];

        }

        iter = iter->next;

    }

    // Создание нового ключа для формирования коллекций

    uint8\_t\* key = (uint8\_t\*)malloc(sizeof(uint8\_t));

    \*key = 1;

    // Добавление результата в выходной список данных

    emitIntermediate(&arg->collection, (void\*)key, (void\*)result, 1);

}

// Функция обработки результатов потоков программы

// Входные данные:

//    KeyValNode\*\* collections;         Массив списков ключей-значений

// Выходные данные:

//    void\*                             Любые данные

void\* merge(KeyValNode\*\* collections, uint8\_t count) {

    // Создание результат типа double\*

    double\* result = (double\*)malloc(sizeof(double));

    \*result = 1;

    // Прохождение по всем спискам ключей-значений

    for (uint8\_t i = 0; i != count; ++i) {

        KeyValNode\* iter = collections[i];

        while (iter) {

            uint8\_t key = \*(uint8\_t\*)iter->key;

            // Слияние всех результатов в один

            if (key == 1) {

                double \*iter\_val = (double\*)iter->val;

                \*result \*= \*iter\_val;

            }

            iter = iter->next;

        }

    }

    return (void\*)result;

}

static double\* arrayCreate(size\_t size) {

    double\* A = (double\*)malloc(sizeof(double) \* size);

    return A;

}

static void arrayRandInit(double\* A, size\_t size) {

    for (size\_t i = 0; i != size; ++i)

        A[i] = rand() % size;

}

static void arrayCopy(double\* dest, double\* src, size\_t size) {

    for (size\_t i = 0; i != size; ++i)

        dest[i] = src[i];

}

static void arryPrint(double\* A, size\_t size) {

    for (size\_t i = 0; i != size; ++i)

        printf("%g ", A[i]);

    printf("\n");

}

static void testResultOutput(FILE\* fp, uint8\_t threads\_count,

                             size\_t array\_size\_min, size\_t array\_size\_max,

                             size\_t measure\_count) {

    fprintf(fp, "size:\tthreads: 1\tthreads: 2\tthreads: 3\tthreads: 4\n");

    for (size\_t size = array\_size\_min; size < array\_size\_max; size \*= 10) {

        double\* A\_src = arrayCreate(size);

        double\* A = arrayCreate(size);

        arrayRandInit(A\_src, size);

        fprintf(fp, "%zu\t", size);

        for (uint8\_t i = 0; i != threads\_count; ++i) {

            arrayCopy(A, A\_src, size);

            double elapsed\_time = 0;

            for (size\_t j = 0; j != measure\_count; ++j) {

                struct timespec start, stop;

                clock\_gettime(CLOCK\_MONOTONIC, &start);

                void\* result = mapReduceChunk((void\*)A, sizeof(double), size,

                                              map, reduce, merge, i + 1);

                clock\_gettime(CLOCK\_MONOTONIC, &stop);

                elapsed\_time += clocktimeDifference(start, stop);

                free(result);

            }

            fprintf(fp, "%lf\t", elapsed\_time / measure\_count);

        }

        fprintf(fp, "\n");

        free(A\_src);

        free(A);

    }

}

static void test(int argc, char\* argv[]) {

    if (argc < TEST\_ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <threads count> <array size min> "

                        "<array size max> <measure count>\n");

        exit(EXIT\_FAILURE);

    }

    srand(time(NULL));

    uint8\_t threads\_count = atoi(argv[1]);

    size\_t array\_size\_min = atoi(argv[2]);

    size\_t array\_size\_max = atoi(argv[3]);

    size\_t measure\_count = atoi(argv[4]);

    FILE\* fp = fopen(RESULT\_FILENAME, "w");

    printf("Program execution...\n");

    testResultOutput(fp, threads\_count, array\_size\_min,

                     array\_size\_max, measure\_count);

    printf("Done.\n");

    fclose(fp);

}

static void demonstration(int argc, char\* argv[]) {

    if (argc < DEMO\_ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <threads count> <array size>\n");

        exit(EXIT\_FAILURE);

    }

    srand(time(NULL));

    uint8\_t threads\_count = atoi(argv[1]);

    size\_t size = atoi(argv[2]);

    double\* A = arrayCreate(size);

    arrayRandInit(A, size);

    printf("Source array:\n");

    arryPrint(A, size);

    void\* result = mapReduceChunk((void\*)A, sizeof(double), size,

                                  map, reduce, merge, threads\_count);

    printf("Transformed array:\n");

    arryPrint(A, size);

    printf("\nMapReduce result:\n");

    if (result)

        printf("%g\n", \*(double\*)result);

    free(result);

    free(A);

}

int main(int argc, char\* argv[]) {

    #ifdef TEST

    test(argc, argv);

    #else

    demonstration(argc, argv);

    #endif

    return 0;

}

*Makefile*

DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE

LIBS    = -lpthread -lm

TARGET  = mapreduce

CF\_DEMO = -std=c99 -O2 -g $(DEFINES)

CF\_TEST = -std=c99 -O2 -g $(DEFINES) -DTEST

demo: CFLAGS = $(CF\_DEMO)

demo: $(TARGET) cleanTemp

test: CFLAGS = $(CF\_TEST)

test: $(TARGET) cleanTemp

$(TARGET): $(TARGET)\_main.o $(TARGET).o

    gcc $(CFLAGS) -o $(TARGET) $(TARGET)\_main.o $(TARGET).o $(LIBS)

$(TARGET)\_main.o: $(TARGET)\_main.c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET)\_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET).c

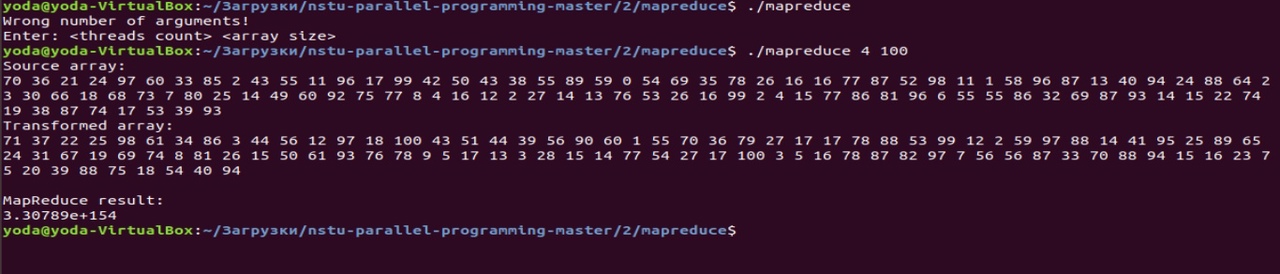
cleanTemp:

    rm -rf \*.o

clean:

    rm -rf $(TARGET)

Результат работы программы:



**example4.cpp**

#include <iostream>

#include <cstdlib>

#include <cstring>

#include <unistd.h>

#include <pthread.h>

#define err\_exit(code, str) {                                       \

    std::cerr << str << ": " << std::strerror(code) << std::endl;   \

    exit(EXIT\_FAILURE);                                             \

}

const size\_t OP\_COUNT    = 100000;

const int    TASKS\_COUNT = 10;

int task\_list[TASKS\_COUNT];

int current\_task = 0;

#ifdef MUTEX

pthread\_mutex\_t mutex;

#endif

void do\_task(int task\_no) {

    for (size\_t i = 0; i != (task\_no + 1) \* OP\_COUNT; ++i);

}

void\* thread\_job(void\* arg) {

    int task\_no;

    int err;

    // Перебираем в цикле доступные задания

    while (true) {

        #ifdef MUTEX

        // Захватываем мьютекс для исключительного доступа

        // к указателю текущего задания (переменная current\_task)

        err = pthread\_mutex\_lock(&mutex);

        if (err != 0)

            err\_exit(err, "Cannot lock mutex");

        #endif

        // Запоминаем номер текущего задания, которое будем исполнять

        task\_no = current\_task;

        sleep(1);

        // Сдвигаем указатель текущего задания на следующее

        current\_task++;

        #ifdef MUTEX

        // Освобождаем мьютекс

        err = pthread\_mutex\_unlock(&mutex);

        if (err != 0)

            err\_exit(err, "Cannot unlock mutex");

        #endif

        // Если запомненный номер задания не превышает

        // количества заданий, вызываем функцию, которая выполнит задание.

        // В противном случае завершаем работу потока

        if (task\_no < TASKS\_COUNT) {

            std::cout << "Thread " << pthread\_self() << " execute the task " << task\_no << std::endl;

            do\_task(task\_no);

        }

        else

            return NULL;

    }

}

int main(int argc, char\* argv[]) {

    pthread\_t thread1, thread2;

    int err;

    #ifdef MUTEX

    // Инициализируем мьютекс

    err = pthread\_mutex\_init(&mutex, NULL);

    if (err != 0)

        err\_exit(err, "Cannot initialize mutex");

    #endif

    // Создаём потоки

    err = pthread\_create(&thread1, NULL, thread\_job, NULL);

    if (err != 0)

        err\_exit(err, "Cannot create thread 1");

    err = pthread\_create(&thread2, NULL, thread\_job, NULL);

    if (err != 0)

        err\_exit(err, "Cannot create thread 2");

    // Дожидаемся завершения потоков

    pthread\_join(thread1, NULL);

    pthread\_join(thread2, NULL);

    #ifdef MUTEX

    // Освобождаем ресурсы, связанные с мьютексом

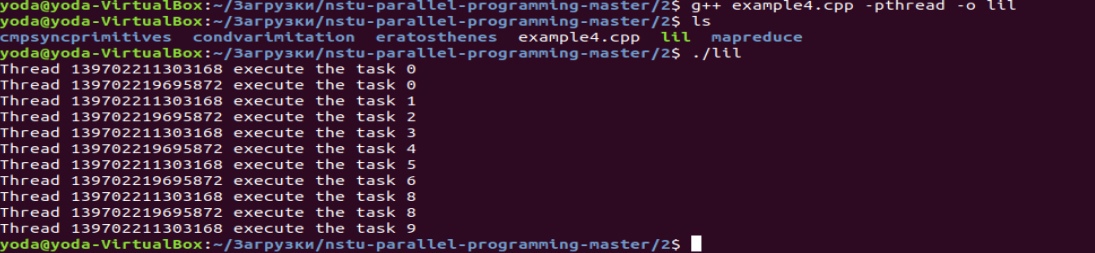
    pthread\_mutex\_destroy(&mutex);

    #endif

    return 0;

}

Результат работы программы



**Задание для самостоятельной работы**

**Eratosthenes**

*eratosthenes.c*

#include "eratosthenes.h"

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

// Функция обработки ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

// Определение размера блока для потока

#define chunkSize(i, chunk\_size, n) \

    (i + chunk\_size < n ? chunk\_size : n - i)

// Статус потока

typedef enum WorkStatus {

    BUSY,                   // Занят

    READY,                  // Готов

    OFF                     // Завершён

} WorkStatus;

// Данные потока

typedef struct Work {

    bool\* data;             // Общие данные

    size\_t begin;           // Индексы блока для обработки

    size\_t end;

    size\_t sieve\_val;       // Число для исключения

    WorkStatus status;      // Статус потока

    pthread\_mutex\_t mutex;

    pthread\_cond\_t cond;

} Work;

// Данные программы

typedef struct Prime {

    uint8\_t threads\_count;  // Кол-во потоков

    size\_t n;               // Кол-во чисел

    size\_t chunk\_size;      // Размер блока для потока

    bool\* data;             // Общие данные (список чисел)

    pthread\_t\* threads;     // Потоки

    struct Work\* works;     // Данные потоков

} Prime;

// Инициализация данных

static void primeDataInit(Prime\* prime, uint8\_t threads\_count, size\_t n, size\_t chunk\_size) {

    prime->threads\_count = threads\_count;

    prime->n = n - FIRST\_VALUE;

    prime->chunk\_size = chunk\_size;

    prime->data = (bool\*)malloc(sizeof(bool) \* prime->n);

    if (prime->data == NULL)

        throwErr("Error: data out of memmory!");

    memset(prime->data, true, sizeof(bool) \* prime->n);

    prime->threads = (pthread\_t\*)malloc(sizeof(pthread\_t) \* prime->threads\_count);

    if (prime->threads == NULL)

        throwErr("Error: threads out of memmory!");

    prime->works = (Work\*)malloc(sizeof(Work) \* prime->threads\_count);

    if (prime->works == NULL)

        throwErr("Error: works out of memmory!");

    // Инициализация потоков для готовности к работе

    int err = 0;

    for (uint8\_t i = 0; i != prime->threads\_count; ++i) {

        prime->works[i].data = prime->data;

        prime->works[i].begin = 0;

        prime->works[i].end = 0;

        prime->works[i].sieve\_val = 0;

        prime->works[i].status = READY;

        err = pthread\_mutex\_init(&prime->works[i].mutex, NULL);

        if (err != 0)

            throwErr("Error: cannot initialize push mutex!");

        err = pthread\_cond\_init(&prime->works[i].cond, NULL);

        if (err != 0)

            throwErr("Error: cannot initialize conditin variable!");

    }

}

// Фильтрация данных потока

static void sieveDataChunk(Work\* work) {

    // Поиск первого кратного простого числа в блоке

    while (work->begin < work->end && (

           work->data[work->begin] == 0 || valFromIndex(work->begin) % work->sieve\_val != 0))

        ++work->begin;

    // Фильтрация данные в блоке с шагом sieve\_val

    for (size\_t i = work->begin; i < work->end; i += work->sieve\_val)

        if (valFromIndex(i) % work->sieve\_val == 0 && valFromIndex(i) != work->sieve\_val)

            work->data[i] = false;

}

// Функция потока

static void\* threadWorker(void\* arg) {

    Work\* work = (Work\*)arg;

    int err = 0;

    // Пока поток не завершён, ждём данные и фильтруем их

    do {

        err = pthread\_mutex\_lock(&work->mutex);

        if (err != 0)

            throwErr("Error: mutex lock!");

        while (work->status == READY) {

            err = pthread\_cond\_wait(&work->cond, &work->mutex);

            if (err != 0)

                throwErr("Error: cannot wait on cond variable!");

        }

        // Если поток занят -- выполняем работу и освобождаем его

        if (work->status == BUSY) {

            sieveDataChunk(work);

            work->status = READY;

        }

        err = pthread\_mutex\_unlock(&work->mutex);

        if (err != 0)

            throwErr("Error: mutex unlock!");

    } while (work->status != OFF);

    pthread\_exit(NULL);

}

// Получение готового к работе потока

static Work\* getReadyThread(Prime\* prime) {

    Work\* work = NULL;

    do {

        bool found = false;

        for (uint8\_t j = 0; j != prime->threads\_count && !found; ++j)

            if (prime->works[j].status == READY) {

                work = &prime->works[j];

                found = true;

            }

    } while (work == NULL);

    return work;

}

// Получение следующего числа для фильтрации

static size\_t getNextSieveVal(Prime\* prime, size\_t cur\_sieve\_val) {

    size\_t sieve\_val = cur\_sieve\_val;

    bool found = false;

    for (size\_t i = sieve\_val - 1; i < prime->n && !found; ++i)

        if (prime->data[i] && sieve\_val < valFromIndex(i)) {

            sieve\_val = valFromIndex(i);

            found = true;

        }

    return sieve\_val;

}

// Завершение работы потоков

static void closeThreads(Prime\* prime) {

    size\_t closed\_count = 0;

    int err = 0;

    // Если тред не занят (готов к работе),

    // присваиваем завершающий статус и посылаем сигнал

    while (closed\_count != prime->threads\_count)

        for (uint8\_t i = 0; i != prime->threads\_count; ++i)

            if (prime->works[i].status == READY) {

                err = pthread\_mutex\_lock(&prime->works[i].mutex);

                if (err != 0)

                    throwErr("Error: mutex lock!");

                ++closed\_count;

                prime->works[i].status = OFF;

                pthread\_cond\_signal(&prime->works[i].cond);

                err = pthread\_mutex\_unlock(&prime->works[i].mutex);

                if (err != 0)

                    throwErr("Error: mutex unlock!");

            }

}

// Создание работы для потоков

static void produceWork(Prime\* prime) {

    size\_t sieve\_val = FIRST\_VALUE;

    int err = 0;

    // Пока число для фильтрации меньше чем sqrt(n)

    while (sieve\_val \* sieve\_val < prime->n) {

        // Делим общие данные на блоки

        for (size\_t i = 0; i < prime->n; i += prime->chunk\_size) {

            // Получаем готовые к работе поток

            Work\* work = getReadyThread(prime);

            err = pthread\_mutex\_lock(&work->mutex);

            if (err != 0)

                throwErr("Error: mutex lock!");

            // Передаём данные потоку и присваиваем стутс "занят"

            work->begin = i;

            work->end = i + chunkSize(i, prime->chunk\_size, prime->n);

            work->sieve\_val = sieve\_val;

            work->status = BUSY;

            pthread\_cond\_signal(&work->cond);

            err = pthread\_mutex\_unlock(&work->mutex);

            if (err != 0)

                throwErr("Error: mutex unlock!");

        }

        // Если все данные, с заданным числом для фильтрации, обработаны

        // Находим новое число для фильтрации

        sieve\_val = getNextSieveVal(prime, sieve\_val);

    }

    // Закрываем потоки

    closeThreads(prime);

}

// Очищаем все данные кроме списка чисел

static void primeDataFree(Prime\* prime) {

    for (uint8\_t i = 0; i != prime->threads\_count; ++i) {

        pthread\_mutex\_destroy(&prime->works[i].mutex);

        pthread\_cond\_destroy(&prime->works[i].cond);

    }

    free(prime->threads);

    free(prime->works);

}

// Поиск простых чисел по: кол-ву потоков, числу n, размеру блока для потока

PrimeNumbers sieveStart(uint8\_t threads\_count, size\_t n, size\_t chunk\_size) {

    Prime prime;

    // Инициализация данных

    primeDataInit(&prime, threads\_count, n, chunk\_size);

    PrimeNumbers prime\_numbers = (PrimeNumbers){prime.data, prime.n};

    // Создание потоков для обработки поступающей работы

    int err = 0;

    for (uint8\_t i = 0; i != prime.threads\_count; ++i) {

        err = pthread\_create(&prime.threads[i], NULL, threadWorker, (void\*)&prime.works[i]);

        if (err != 0)

            throwErr("Error: cannot create a thread!");

    }

    // Создание работы для потоков

    produceWork(&prime);

    for (uint8\_t i = 0; i != prime.threads\_count; ++i) {

        err = pthread\_join(prime.threads[i], NULL);

         if (err != 0)

            throwErr("Error: cannot join a thread!");

    }

    primeDataFree(&prime);

    return prime\_numbers;

}

*eratosthenes.h*

#ifndef ERATOSTHENES\_H

#define ERATOSTHENES\_H

#include <stdio.h>

#include <stdint.h>

#include <stdbool.h>

// Первое простое число в списке

#define FIRST\_VALUE 2

// Список простых чисел

typedef struct PrimeNumbers {

    bool\* data;             // Статус числа в списке

    size\_t n;               // Кол-во чисел списка

} PrimeNumbers;

// Получить число из списка по индексу

static inline size\_t valFromIndex(size\_t i) {

    return i + FIRST\_VALUE;

}

// Вывод простых чисел на экран

static inline void printPrimeNumbers(const PrimeNumbers\* prime\_numbers) {

    for (size\_t i = 0; i != prime\_numbers->n; ++i)

        if (prime\_numbers->data[i])

            printf("%zu ", valFromIndex(i));

    printf("\n");

}

// Поиск простых чисел по: кол-ву потоков, числу n, размеру блока для потока

PrimeNumbers sieveStart(uint8\_t threads\_count, size\_t n, size\_t chunk\_size);

#endif

*eratosthenes\_main.c*

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include "eratosthenes.h"

#define DEMO\_ARGS\_COUNT 4

#define TEST\_ARGS\_COUNT 5

#define RESULT\_FILENAME "result.txt"

#define BILLION 1.0E+9

// Функция вычета разности между временными величинами

#define clocktimeDifference(start, stop)            \

    1.0 \* (stop.tv\_sec - start.tv\_sec) +            \

    1.0 \* (stop.tv\_nsec - start.tv\_nsec) / BILLION

static void testResultOutput(FILE\* fp, uint8\_t threads\_count,

                             size\_t array\_size\_min, size\_t array\_size\_max,

                             size\_t measure\_count) {

    fprintf(fp, "size:\tthreads: 1\tthreads: 2\tthreads: 3\tthreads: 4\n");

    for (size\_t size = array\_size\_min; size < array\_size\_max; size \*= 10) {

        fprintf(fp, "%zu\t", size);

        for (uint8\_t i = 0; i != threads\_count; ++i) {

            double elapsed\_time = 0;

            for (size\_t j = 0; j != measure\_count; ++j) {

                struct timespec start, stop;

                clock\_gettime(CLOCK\_MONOTONIC, &start);

                PrimeNumbers prime\_numbers = sieveStart(i + 1, size, size / 10);

                clock\_gettime(CLOCK\_MONOTONIC, &stop);

                elapsed\_time += clocktimeDifference(start, stop);

                free(prime\_numbers.data);

            }

            fprintf(fp, "%lf\t", elapsed\_time / measure\_count);

        }

        fprintf(fp, "\n");

    }

}

static void test(int argc, char\* argv[]) {

    if (argc < TEST\_ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <threads count> <array size min> "

                        "<array size max> <measure count>\n");

        exit(EXIT\_FAILURE);

    }

    srand(time(NULL));

    uint8\_t threads\_count = atoi(argv[1]);

    size\_t array\_size\_min = atoi(argv[2]);

    size\_t array\_size\_max = atoi(argv[3]);

    size\_t measure\_count = atoi(argv[4]);

    FILE\* fp = fopen(RESULT\_FILENAME, "w");

    printf("Program execution...\n");

    testResultOutput(fp, threads\_count, array\_size\_min, array\_size\_max, measure\_count);

    printf("Done.\n");

    fclose(fp);

}

static void demonstration(int argc, char\* argv[]) {

    if (argc < DEMO\_ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <threads count> <n limit> <chunk size>\n");

        exit(EXIT\_FAILURE);

    }

    uint8\_t threads\_count = atoi(argv[1]);

    size\_t n = atol(argv[2]);

    size\_t chunk\_size = atol(argv[3]);

    struct timespec start, stop;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    PrimeNumbers prime\_numbers = sieveStart(threads\_count, n, chunk\_size);

    clock\_gettime(CLOCK\_MONOTONIC, &stop);

    double elapsed\_time = clocktimeDifference(start, stop);

    printPrimeNumbers(&prime\_numbers);

    printf("\n\nElapsed time: %lf\n", elapsed\_time);

    free(prime\_numbers.data);

}

int main(int argc, char\* argv[]) {

    #ifdef TEST

    test(argc, argv);

    #else

    demonstration(argc, argv);

    #endif

    return 0;

}

*Makefile*

DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE

LIBS    = -lpthread -lm

TARGET  = eratosthenes

CF\_DEMO = -std=c99 -O2 -g $(DEFINES)

CF\_TEST = -std=c99 -O2 -g $(DEFINES) -DTEST

demo: CFLAGS = $(CF\_DEMO)

demo: $(TARGET) cleanTemp

test: CFLAGS = $(CF\_TEST)

test: $(TARGET) cleanTemp

$(TARGET): $(TARGET)\_main.o $(TARGET).o

    gcc $(CFLAGS) -o $(TARGET) $(TARGET)\_main.o $(TARGET).o $(LIBS)

$(TARGET)\_main.o: $(TARGET)\_main.c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET)\_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET).c

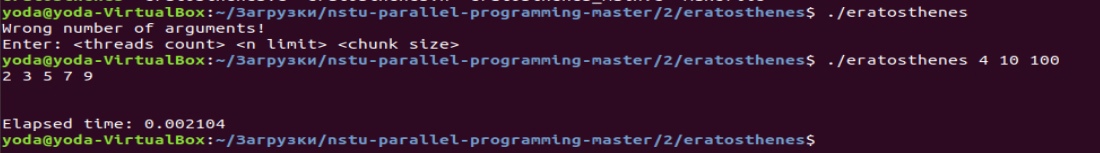
cleanTemp:

    rm -rf \*.o

clean:

    rm -rf $(TARGET)

Результат работы программы:



Вывод

Мы познакомились со средствами планирования и синхронизации потоков, научились работать с потоками, которые обмениваются информацией между собой.