Работа над ошибками 1 задания лабораторной работы 4.

```
@Egor228zorro ➜/workspaces/os_lab_2019/lab4/src (master) $ ./parallel_min_max --seed 50 --array_size 1000 --pnum 4 --timeout 10
Истек тайм-аут. Завершение дочерних процессов.
Min: 2261774
Max: 2137753767
Затраченное время: 10000.596000ms
```

```
#include <ctype.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>//нужна для работы с сигналами

#include <getopt.h>

#include "find_min_max.h"
#include "utils.h"

pid_t child_pids[100];
int pnum_global;

void timeout_handler(int sig) {
    printf("Истек тайм-аут. Завершение дочерних процессов.\n");
    for (int i = 0; i < pnum_global; i++) {
        if (child_pids[i] > 0) {
            kill(child_pids[i], SIGKILL);
        }
```

```
    }
}

int main(int argc, char **argv) {
    int seed = -1;          // Нач значение для генератора случайных чисел (инициализируется -1 для проверки)
    int array_size = -1;    // Размер массива (инициализируется -1 для проверки)
    int pnum = -1;          // Количество дочерних процессов (инициализируется -1 для проверки)
    bool with_files = false; // Флаг, указывающий, использовать ли файлы для передачи данных (по умолчанию false)
    int timeout = -1;       // Время ожидания завершения дочерних процессов в секундах (инициализируется -1 для проверки)

    // Разбор аргументов командной строки с использованием getopt_long
    while (true) {
        int current_optind = optind ? optind : 1; // Сохраняем текущий индекс optind

        // Определение структуры для описания опций командной строки
        static struct option options[] = {
            {"seed", required_argument, 0, 0},       // Опция --seed с обязательным аргументом
            {"array_size", required_argument, 0, 0}, // Опция --array_size с обязательным аргументом
            {"pnum", required_argument, 0, 0},       // Опция --pnum с обязательным аргументом
            {"by_files", no_argument, 0, 'f'},       // Опция --by_files без аргумента (короткая форма -f)
            {"timeout", required_argument, 0, 0},    // Опция --timeout с обязательным аргументом
            {0, 0, 0, 0}                             // Нулевой элемент, обозначающий конец массива опций
        };
```

```
        int option_index = 0;                           // Индекс текущей опции
        int c = getopt_long(argc, argv, "f", options, &option_index); // Разбор опции

        if (c == -1) break; // Если getopt_long вернул -1, значит, опций больше нет

        switch (c) {
            case 0: // Обработка длинных опций (например, --seed, --array_size)
                switch (option_index) {
                    case 0: // --seed
                        seed = atoi(optarg); // Преобразование аргумента в целое число
                        break;
                    case 1: // --array_size
                        array_size = atoi(optarg); // Преобразование аргумента в целое число
                        break;
                    case 2: // --pnum
                        pnum = atoi(optarg); // Преобразование аргумента в целое число
                        break;
                    case 3: // --by_files
                        with_files = true; // Установка флага использования файлов
                        break;
                    case 4: // --timeout
                        timeout = atoi(optarg); // Преобразование аргумента в целое число
                        break;

                    default:
                        printf("Index %d is out of options\n", option_index); // Обработка
неизвестного индекса
                }
                break;
            case 'f': // Обработка короткой опции -f (--by_files)
                with_files = true; // Установка флага использования файлов
```

```
            break;


        case '?': // Обработка неизвестной опции
            break;


        default:
            printf("getopt возвращает код символа 0%o?\n", c); // Обработка неожиданного возвращаемого значения getopt
    }
}


// Проверка наличия аргументов, не являющихся опциями
if (optind < argc) {
    printf("Имеет по крайней мере один аргумент без опции\n");
    return 1;
}


// Проверка, что все необходимые аргументы заданы
if (seed == -1 || array_size == -1 || pnum == -1) {
    printf("Usage: %s --seed \"num\" --array_size \"num\" --pnum \"num\" [--timeout \"num\"] \n", argv[0]);
    return 1;
}


// Выделение памяти для массива
int *array = malloc(sizeof(int) * array_size);
if (array == NULL) {
    perror("malloc"); // Вывод сообщения об ошибке, если не удалось выделить память
    return 1;
}
```

```
    // Заполнение массива случайными числами
    GenerateArray(array, array_size, seed);

    int active_child_processes = 0; // Счетчик активных дочерних процессов
    struct timeval start_time;       // Структура для хранения времени начала
    gettimeofday(&start_time, NULL);  // Получение текущего времени

    // Создание pipe для передачи данных от дочерних процессов
    int pipes[pnum][2]; // Массив pipe для каждого дочернего процесса
    for (int i = 0; i < pnum; i++) {
        if (pipe(pipes[i]) < 0) { // Создание pipe
            perror("pipe");        // Вывод сообщения об ошибке, если не удалось создать
pipe
            free(array);           // Освобождение выделенной памяти
            return 1;
        }
    }

    pnum_global = pnum;

    // Создание дочерних процессов
    for (int i = 0; i < pnum; i++) {
        pid_t child_pid = fork(); // Создание дочернего процесса
        child_pids[i] = child_pid;  // Сохраняем PID
        if (child_pid >= 0) {      // Проверка, успешно ли создан дочерний процесс
            active_child_processes += 1; // Увеличение счетчика активных процессов
            if (child_pid == 0) {      // Код дочернего процесса
                close(pipes[i][0]); // Закрыть чтение (не используется в дочернем
процессе)
```

```
        int start_idx = (array_size / pnum) * i;                    // Вычисление начального индекса для текущего процесса
        int end_idx = (i == pnum - 1) ? array_size : (array_size / pnum) * (i + 1); // Вычисление конечного индекса

        // Вычисление минимального и максимального значений в заданном диапазоне
        struct MinMax min_max = GetMinMax(array, start_idx, end_idx);

        if (with_files) { // Если используется передача данных через файлы
            char filename[20];                                      // Имя файла
            sprintf(filename, "result_%d.txt", i);                  // Формирование имени файла
            FILE *file = fopen(filename, "w");                      // Открытие файла для записи
            if (file) {                                             // Проверка, успешно ли открыт файл
                fprintf(file, "min: %d\nmax: %d\n", min_max.min, min_max.max); // Запись минимального и максимального значений в файл
                fclose(file);                                       // Закрытие файла
            } else {
                perror("fopen"); // Вывод сообщения об ошибке, если не удалось открыть файл
            }
        } else {                                                    // Если используется передача данных через pipe
            write(pipes[i][1], &min_max, sizeof(min_max)); // Запись структуры MinMax в pipe
        }

        close(pipes[i][1]); // Закрыть запись (больше не используется)
        sleep(timeout); // <--- ДОБАВЛЕННЫЙ ВЫЗОВ sleep ДЛЯ ИМИТАЦИИ ЗАДЕРЖКИ
        free(array);        // Освобождение выделенной памяти
        exit(0);            // Завершение дочернего процесса
```

```
            }
        } else {
            perror("fork"); // Вывод сообщения об ошибке, если не удалось создать дочерний процесс
            free(array);    // Освобождение выделенной памяти
            return 1;
        }
    }

    if (timeout > 0) {
        signal(SIGALRM, timeout_handler);
        alarm(timeout);
    }

    int finished_processes = 0;
    while (finished_processes < pnum) {
        for (int i = 0; i < pnum; i++) {
            if (child_pids[i] <= 0) continue;

            int status;
            pid_t result = waitpid(child_pids[i], &status, WNOHANG);
            if (result > 0) {
                active_child_processes -= 1;
                finished_processes++;
                child_pids[i] = 0;
            } else if (result < 0) {
                perror("waitpid");
            }
        }
    }
```

```
    // Сбор результатов
    struct MinMax min_max; // Структура для хранения общего минимального и максимального значений
    min_max.min = INT_MAX;  // Инициализация минимального значения максимальным возможным значением int
    min_max.max = INT_MIN;  // Инициализация максимального значения минимальным возможным значением int

    for (int i = 0; i < pnum; i++) { // Перебор всех дочерних процессов
        int min = INT_MAX;          // Локальное минимальное значение
        int max = INT_MIN;          // Локальное максимальное значение

        if (with_files) { // Если используется передача данных через файлы
            char filename[20];                              // Имя файла
            sprintf(filename, "result_%d.txt", i);          // Формирование имени файла
            FILE *file = fopen(filename, "r");              // Открытие файла для чтения
            if (file) {                                     // Проверка, успешно ли открыт файл
                fscanf(file, "min: %d\nmax: %d\n", &min, &max); // Чтение минимального и максимального значений из файла
                fclose(file);                               // Закрытие файла
            } else {
                perror("fopen"); // Вывод сообщения об ошибке, если не удалось открыть файл
            }
        } else { // Если используется передача данных через pipe
            close(pipes[i][1]); // Закрыть запись (не используется в родительском процессе)
            ssize_t bytes_read = read(pipes[i][0], &min_max, sizeof(min_max)); // Чтение структуры MinMax из pipe
            close(pipes[i][0]); // Закрыть чтение
            if (bytes_read > 0) { // Проверка, удалось ли прочитать данные из pipe
                min = min_max.min; // Извлечение минимального значения
                max = min_max.max; // Извлечение максимального значения
```

```
        } else {
            // Если не удалось прочитать данные из pipe, устанавливаем значения по умолчанию
            min = INT_MAX;
            max = INT_MIN;
        }
    }

    if (min < min_max.min) min_max.min = min; // Обновление общего минимального значения
    if (max > min_max.max) min_max.max = max; // Обновление общего максимального значения
  }

  struct timeval finish_time;       // Структура для хранения времени окончания
  gettimeofday(&finish_time, NULL);  // Получение текущего времени

  double elapsed_time = (finish_time.tv_sec - start_time.tv_sec) * 1000.0; // Вычисление разницы во времени в миллисекундах
  elapsed_time += (finish_time.tv_usec - start_time.tv_usec) / 1000.0;    // Добавление микросекунд

  free(array); // Освобождение выделенной памяти

  printf("Min: %d\n", min_max.min);                          // Вывод минимального значения
  printf("Max: %d\n", min_max.max);                          // Вывод максимального значения
  printf("Затраченное время: %fms\n", elapsed_time); // Вывод затраченного времени
  return 0;
}
```