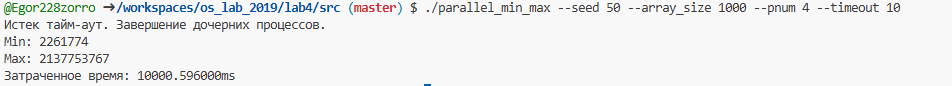
**Работа над ошибками 1 задания лабораторной работы 4.**  


#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;

}