Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №2 по курсу

«Операционные системы»

Группа: М8О-211БВ-24

Студент: Николич С.Р.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 30.10.25

Москва, 2025

Постановка задачи

Вариант 3.

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.
В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант 3) Отсортировать массив целых чисел при помощи параллельной сортировки слиянием.

Общий метод и алгоритм решения

Алгоритм параллельной сортировки слиянием:

1. Исходный массив рекурсивно разделяется на подмассивы
2. Для обработки левой и правой половин создаются отдельные потоки, если не превышен лимит потоков
3. Подмассивы размером менее 1000 элементов сортируются последовательно
4. Отсортированные подмассивы сливаются в единый массив

Синхронизация:

- Глобальный счетчик active_threads ограничивает число одновременно работающих потоков
- Мьютекс защищает доступ к счетчику потоков
- Для ожидания завершения потоков используется pthread_join

Использованные системные вызовы

- pthread_t - тип для идентификатора потока
- pthread_create() - создает новый поток
- pthread_join() - ожидает завершения указанного потока
- pthread_mutex_t - тип для мьютекса
- pthread_mutex_init() - инициализирует мьютекс
- pthread_mutex_lock() - захватывает мьютекс
- pthread_mutex_unlock() - освобождает мьютекс
- malloc() - выделяет память
- free() - освобождает память
- gettimeofday() - получает текущее время

- sysconf(_SC_NPROCESSORS_ONLN) - получает количество доступных процессоров

В рамках лабораторной работы программа реализует многопоточную сортировку массива с использованием алгоритма слияния. Родительский поток создает дочерние потоки для параллельной обработки частей массива. Для ограничения количества одновременно работающих потоков используется глобальный счетчик, защищенный мьютексом.

Когда размер подмассива становится менее 1000 элементов, применяется последовательная сортировка для уменьшения накладных расходов. Отсортированные части массива сливаются в единый отсортированный массив.

Программа измеряет время последовательной и параллельной сортировки, вычисляет ускорение и эффективность алгоритма. Когда пользователь завершает ввод, программа выводит результаты исследования зависимости производительности от количества потоков.

Код программы

parallel.c

```c
#include <stdint.h>
#include <stdbool.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <string.h>
#include <sys/time.h>
#include "parallel.h"

typedef struct {
    int *array;
    int left;
    int right;
    int depth;
    int max_threads;
} SortArgs;

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int active_threads = 0;

//Функция для получения времени
double get_time(){
    struct timeval tv;
    gettimeofday(&tv, NULL);//Системный вызов для получения текущего времени
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

// Последовательное слияние двух отсортированных массивов
void merge(int *array, int left, int mid, int right){

    int n1 = mid - left + 1;// Размер левого подмассива
    int n2 = right - mid;// Правого
```

```
    int *left_arr = malloc(n1 * sizeof(int));
    int *right_arr = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++){
        left_arr[i] = array[left + i];
    }
    for (int j = 0; j < n2; j++){
        right_arr[j] = array[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2){
        if (left_arr[i] <= right_arr[j]){
            array[k] = left_arr[i];
            i++;
        }
        else{
            array[k] = right_arr[j];
            j++;
        }
        k++;
    }
    //Копируем оставшиеся элементы из левого массива, на случай
    //если правый закончился раньше
    while (i < n1){
        array[k] = left_arr[i];
        i++;
        k++;
    }

    while (j < n2){
        array[k] = right_arr[j];
        j++;
        k++;
    }

    free(left_arr);
    free(right_arr);
}

// Последовательная сортировка слиянием
void sequential_merge_sort(int *array, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        sequential_merge_sort(array, left, mid);
        sequential_merge_sort(array, mid + 1, right);

        merge(array, left, mid, right);
    }
}

// Функция, выполняемая в потоке для сортировки
static void *parallel_merge_sort_thread(void *_args){
```

```
//Явное преобразования типа
SortArgs *args = (SortArgs *)_args;
int *array = args->array;
int left = args->left;
int right = args->right;
int depth = args->depth;
int max_threads = args->max_threads;

if (left >= right || (right - left) < 1000){
    // условия выхода из рекурсии:
    // 1. left >= right - массив пустой или из одного элемента
    // 2. (right - left) < 1000 - массив достаточно маленький используем
    // Последовательную сортировку
    sequential_merge_sort(array, left, right);
    free(args);
    return NULL;
}

int mid = left + (right - left) / 2;

SortArgs *left_args = malloc(sizeof(SortArgs));
SortArgs *right_args = malloc(sizeof(SortArgs));

left_args->array = array;
left_args->left = left;
left_args->right = mid;
left_args->depth = depth + 1;
left_args->max_threads = max_threads;

right_args->array = array;
right_args->left = mid + 1;
right_args->right = right;
right_args->depth = depth + 1;
right_args->max_threads = max_threads;

pthread_t left_thread; // Идентификатор потока для левой половины
int left_created = 0; // Флаг создания потока (0/1)

pthread_mutex_lock(&mutex);
if (active_threads < max_threads){
    active_threads++;
    left_created = 1;
}
pthread_mutex_unlock(&mutex);

if (left_created){
    pthread_create(&left_thread, NULL, parallel_merge_sort_thread, left_args);
    //NULL - атрибуты потока по умолчанию
}
else{
    //Лимит потоков исчерпан, вызываем рекурсию
    parallel_merge_sort_thread(left_args);
}
```

```
    //Правая половина всегда обрабатывается в текущем потоке
    parallel_merge_sort_thread(right_args);

    if (left_created){
        pthread_join(left_thread, NULL); // Ожидание потока
        pthread_mutex_lock(&mutex);
        active_threads--;
        pthread_mutex_unlock(&mutex);
    }

    merge(array, left, mid, right);
    free(args);
    return NULL;
}

// Основная функция параллельной сортировки слиянием
void parallel_merge_sort(int *array, int size, int max_threads) {
    SortArgs *args = malloc(sizeof(SortArgs));
    args->array = array;
    args->left = 0;
    args->right = size - 1;
    args->depth = 0;
    args->max_threads = max_threads;

    parallel_merge_sort_thread(args);
}

int is_sorted(int *array, int size) {
    for (int i = 1; i < size; i++) {
        if (array[i] < array[i - 1]) {
            return 0;
        }
    }
    return 1;
}

void print_array(int *array, int size) {
    printf("[");
    for (int i = 0; i < size && i < 20; i++) {
        printf("%d", array[i]);
        if (i < size - 1 && i < 19) printf(", ");
    }
    if (size > 20) printf(", ...");
    printf("]\n");
}
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "parallel.h"
```

```c
int main(int argc, char **argv){
    if (argc < 3) {
        printf("Использование: %s <размер_массива> <макс_потоков> [--print]\n", argv[0]);
        return 1;
    }

    int size = atoi(argv[1]);
    int max_threads = atoi(argv[2]);
    int print_array_flag = (argc > 3 && strcmp(argv[3], "--print") == 0);

    if (size <= 0 || max_threads <= 0){
        printf("Ошибка: размер массива и количество потоков должны быть
положительными\n");
        return 1;
    }

    int *array = malloc(size * sizeof(int));
    int *array_seq = malloc(size * sizeof(int)); // Для последовательной сортировки
    int *array_test = malloc(size * sizeof(int)); // Копия для тестов

    srand(time(NULL)); // Инициализируем генератор случайных чисел
    for (int i = 0; i < size; i++){
        array[i] = rand() % 1000;
        array_seq[i] = array[i];
        array_test[i] = array[i];
    }

    if (print_array_flag){
        printf("Исходный массив: ");
        print_array(array, size);
    }

    printf("Размер массива: %d\n", size);
    printf("Максимальное количество потоков: %d\n", max_threads);
    printf("Количество ядер в системе: %ld\n", sysconf(_SC_NPROCESSORS_ONLN));

    double start_time = get_time();
    parallel_merge_sort(array, size, max_threads);
    double parallel_time = get_time() - start_time;

    start_time = get_time();
    sequential_merge_sort(array_seq, 0, size - 1);
    double sequential_time = get_time() - start_time;

    if (!is_sorted(array, size)){
        printf("ОШИБКА: Массив не отсортирован после параллельной сортировки!\n");
        free(array);
        free(array_seq);
        free(array_test);
        return 1;
    }

    if (!is_sorted(array_seq, size)){
        printf("ОШИБКА: Массив не отсортирован после последовательной сортировки!\n");
```

```
        free(array);
        free(array_seq);
        free(array_test);
        return 1;
    }

    if (print_array_flag){
        printf("Отсортированный массив: ");
        print_array(array, size);
    }

    double speedup = sequential_time / parallel_time;
    double efficiency = speedup / max_threads;

    printf("\n=== РЕЗУЛЬТАТЫ ===\n");
    printf("Время последовательной сортировки: %.6f сек\n", sequential_time);
    printf("Время параллельной сортировки: %.6f сек\n", parallel_time);
    printf("Ускорение (Speedup): %.4f\n", speedup);
    printf("Эффективность (Efficiency): %.4f\n", efficiency);

    printf("\n=== ИССЛЕДОВАНИЕ ЗАВИСИМОСТИ ОТ КОЛИЧЕСТВА ПОТОКОВ ===\n");
    printf("Потоки\tВремя(с)\tУскорение\tЭффективность\n");
    printf("-------------------------------------------------\n");

    for (int threads = 1; threads <= max_threads; threads++) {
        memcpy(array, array_test, size * sizeof(int));

        start_time = get_time();
        parallel_merge_sort(array, size, threads);
        double time = get_time() - start_time;

        double sp = sequential_time / time;
        double eff = sp / threads;

        printf("%d\t%.6f\t%.4f\t%.4f\n", threads, time, sp, eff);
    }

    free(array);
    free(array_seq);
    free(array_test);

    return 0;
}
```

parallel.h

```
#ifndef PARALLEL_H
#define PARALLEL_H

#include <pthread.h>

double get_time(void);
void merge(int *array, int left, int mid, int right);
```

```
void sequential_merge_sort(int *array, int left, int right);
void parallel_merge_sort(int *array, int size, int max_threads);
int is_sorted(int *array, int size);
void print_array(int *array, int size);

#endif
```

Протокол работы программы

```
savva@Honorlaptop:~/OS/lab2$ ./main 10000000 7 --print
Исходный массив: [960, 989, 46, 22, 569, 226, 510, 251, 378, 198, 574, 458, 858, 78, 362, 873, 582, 780, 1, 104, ...]
Размер массива: 10000000
Максимальное количество потоков: 7
Количество ядер в системе: 16
Отсортированный массив: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...]

=== РЕЗУЛЬТАТЫ ===
Время последовательной сортировки: 1.309905 сек
Время параллельной сортировки: 0.328079 сек
Ускорение (Speedup): 3.9927
Эффективность (Efficiency): 0.5704

=== ИССЛЕДОВАНИЕ ЗАВИСИМОСТИ ОТ КОЛИЧЕСТВА ПОТОКОВ ===
Потоки  Время(с)        Ускорение       Эффективность
------------------------------------------------
1       0.813297        1.6106  1.6106
2       0.585566        2.2370  1.1185
3       0.425753        3.0767  1.0256
4       0.395120        3.3152  0.8288
5       0.345039        3.7964  0.7593
6       0.301692        4.3419  0.7236
7       0.274491        4.7721  0.6817
savva@Honorlaptop:~/OS/lab2$
```

Вывод

В данной лабораторной работе я освоил управление потоками в операционной системе и организацию синхронизации между ними. Была реализована многопоточная архитектура, где основной поток распределяет данные между рабочими потоками через общую память с использованием мьютексов для синхронизации. Каждый рабочий поток выполнял независимую обработку частей массива, что позволило организовать параллельное выполнение сортировки. В ходе работы я научился работать с функциями для управления потоками, мьютексами и разделяемыми данными, а также освоил методы измерения производительности и анализа эффективности параллельных алгоритмов.