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-210БВ-24

Студент: Тодуа Н.Г.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 01.10.25

Москва, 2025

Постановка задачи

Вариант 4.

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.

Отсортировать массив целых чисел при помощи TimSort

Общий метод и алгоритм решения

Использованные системные вызовы:

- clock_gettime() - измерение времени
- pthread_create() - создание потока
- pthread_join() - ожидание завершения потока. Блокирует вызывающий поток до завершения указанного потока

Параллельная сортировка реализована с использованием **POSIX Threads.** Основной процесс разбивает исходный массив на участки фиксированного размера и создаёт несколько потоков для одновременного выполнения сортировки вставками на каждом участке. После завершения всех потоков основной процесс последовательно объединяет отсортированные участки с помощью процедуры слияния. Измерение времени выполнения проводится с использованием высокоточного таймера clock_gettime. Программа позволяет оценить эффективность параллельной реализации по сравнению с последовательной за счёт расчёта ускорения и эффективности при различном числе потоков.

Код программы

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

#define ARRAY_SIZE 100000
#define RUN_SIZE 10000

typedef struct {
    int * arr;
    int left;
    int right;
} ThreadArgs;

void insertion_sort(int * arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void merge(int * arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int * L = (int *)malloc(size: n1 * sizeof(int));
    int * R = (int *)malloc(size: n2 * sizeof(int));

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
```

```
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(ptr: L);
    free(ptr: R);
}

void merge_all(int * arr, int n, int run_size) {
    for (int size = run_size; size < n; size *= 2) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = left + size - 1;
            int right = left + 2 * size - 1;
            if (mid >= n - 1) continue;
            if (right >= n) right = n - 1;
            merge(arr, left, mid, right);
        }
    }
}

void * thread_sort(void * param) {
    ThreadArgs * args = (ThreadArgs *)param;
    insertion_sort(arr: args->arr, left: args->left, right: args->right);
    free(ptr: args);
    return NULL;
}

void generate_array(int * arr, int n) {
    for (int i = 0; i < n; i++) arr[i] = rand() % 100000;
}

void copy_array(int * src, int * dst, int n) {
    for (int i = 0; i < n; i++) dst[i] = src[i];
}
```

```c
double get_time_ms() {
    struct timespec ts;
    clock_gettime(clock_id: CLOCK_MONOTONIC, tp: &ts);
    return (double)ts.tv_sec * 1000.0 + (double)ts.tv_nsec / 1000000.0;
}

double sequential_sort(int * arr, int n) {
    double start = get_time_ms();
    for (int i = 0; i < n; i += RUN_SIZE) {
        int left = i;
        int right = (i + RUN_SIZE - 1 < n) ? i + RUN_SIZE - 1 : n - 1;
        insertion_sort(arr, left, right);
    }
    merge_all(arr, n, run_size: RUN_SIZE);
    double end = get_time_ms();
    return end - start;
}

double parallel_sort(int * arr, int n, int max_threads) {
    int run_count = (n + RUN_SIZE - 1) / RUN_SIZE;
    double start = get_time_ms();
    int current = 0;

    while (current < run_count) {
        int active = 0;
        pthread_t * threads = (pthread_t *)malloc(size: max_threads * sizeof(pthread_t));

        for (; active < max_threads && current < run_count; active++, current++) {
            int left = current * RUN_SIZE;
            int right = (left + RUN_SIZE - 1 < n) ? left + RUN_SIZE - 1 : n - 1;

            ThreadArgs * arg = (ThreadArgs *)malloc(size: sizeof(ThreadArgs));
            arg->arr = arr;
            arg->left = left;
            arg->right = right;

            if (pthread_create(newthread: &threads[active], attr: NULL, start_routine: thread_sort, arg) != 0) {
                perror(s: "Ошибка в создании потока");
                exit(status: 1);
            }
```

```
        }

        for (int i = 0; i < active; i++) {
            pthread_join(th: threads[i], thread_return: NULL);
        }
        free(ptr: threads);
    }

    merge_all(arr, n, run_size: RUN_SIZE);
    double end = get_time_ms();
    return end - start;
}

int main() {
    srand(seed: (unsigned)time(timer: NULL));

    int * arr_original = (int *)malloc(size: ARRAY_SIZE * sizeof(int));
    int * arr_copy = (int *)malloc(size: ARRAY_SIZE * sizeof(int));
    generate_array(arr: arr_original, n: ARRAY_SIZE);

    printf(format: "Размер массива: %d\n", ARRAY_SIZE);

    copy_array(src: arr_original, dst: arr_copy, n: ARRAY_SIZE);
    double t_seq = sequential_sort(arr: arr_copy, n: ARRAY_SIZE);
    printf(format: "Последовательная: %.3f мс\n", t_seq);

    int threads_to_test[] = {[0]=1, [1]=2, [2]=4, [3]=8, [4]=16, [5]=32, [6]=64, [7]=128};
    int num_tests = sizeof(threads_to_test) / sizeof(threads_to_test[0]);

    printf(format: "\n Потоки | Время | Ускорение | Эффективность\n");
    printf(format: "----------------------------------------\n");

    for (int i = 0; i < num_tests; i++) {
        int th = threads_to_test[i];
        copy_array(src: arr_original, dst: arr_copy, n: ARRAY_SIZE);
        double t_par = parallel_sort(arr: arr_copy, n: ARRAY_SIZE, max_threads: th);
        double speedup = t_seq / t_par;
        double eff = (speedup / th) * 100.0;
        printf(format: "%7d | %9.3f | %7.3f | %10.2f%%\n", th, t_par, speedup, eff);
    }
```

```
    free(ptr: arr_original);
    free(ptr: arr_copy);
    return 0;
}
```

Протокол работы программы

Тестирование:

```
[mafonchik@mafonchik-aspirea31544p output]$ ./"main"
Размер массива: 100000
Последовательная: 1464.058 мс

 Потоки | Время | Ускорение | Эффективность
----------------------------------------
      1 |  1479.962 |    0.989 |      98.93%
      2 |   752.669 |    1.945 |      97.26%
      4 |   466.612 |    3.138 |      78.44%
      8 |   321.901 |    4.548 |      56.85%
     16 |   245.552 |    5.962 |      37.26%
     32 |   256.656 |    5.704 |      17.83%
     64 |   278.562 |    5.256 |       8.21%
    128 |   258.836 |    5.656 |       4.42%
[mafonchik@mafonchik-aspirea31544p output]$
```

Вывод

Программа демонстрирует ускорение при использовании многопоточности, однако эффективность резко снижается после определённого количества потоков. Наибольшее ускорение (~5.96×) достигается при 16 потоках, что совпадает количеством логических процессоров в системе – это ожидаемо, так как каждому потоку в этот момент может быть выделено отдельное аппаратное ядро/поток без избыточного переключения контекста. При использовании меньшего числа потоков (2-8) наблюдается хорошая масштабируемость: ускорение близко к линейному, а эффективность остаётся высокой (от 98% до 56%). Однако при превышении числа логических процессоров (32 и более потоков) ускорение не только перестаёт расти, но и снижается, а эффективность значительно падает. Это связано с накладными расходами на создание и управление избыточным количеством потоков, конкуренцию за память и кэш, а также частыми переключениями контекста, которые перевешивают выгоду от параллелизма

Число потоков	Время исполнения (мс)	Ускорение	Эффективность
1	1479.962	0.989	98.93%
2	752.669	1.945	97.26%
4	466.612	3.138	78.44%
8	321.901	4.548	56.85%
16	245.552	5.962	37.26%
32	256.656	5.704	17.83%
64	278.562	5.256	8.21%
128	258.836	5.656	4.42%

.

Таким образом, оптимальное число потоков для данной задачи и аппаратной конфигурации – 16, что соответствует количеству логических процессоров. Дальнейшее увеличение числа потоков бессмысленно и приводит к падению производительности.

В отчёте использованы две ключевые метрики параллельных вычислений: ускорение (speedup) и эффективность (efficiency). Ускорение показывает, во сколько раз параллельная реализация быстрее последовательной, и вычисляется по формуле:

$$Speedup = T_{SEQ}/T_{PAR}$$

где Tseq – время выполнения последовательного алгоритма, а Tpar – время выполнения параллельного варианта при заданном числе потоков. Эффективность оценивает, насколько рационально используются вычислительные ресурсы, и рассчитывается как

$$Efficiency = (Speedup / p) * 100\%$$

где p – количество используемых потоков. Значение эффективности близкое к 100% означает почти идеальное масштабирование, тогда как её снижение указывает на рост накладных расходов и/или недостаточную параллелизуемость задачи.