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: Мицкевич А.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 30.10.25

Москва, 2025

Постановка задачи

Вариант 4.

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке 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:

- DWORD WINAPI WaitForMultipleObjects(DWORD nCount, const HANDLE *lpHandles, BOOL bWaitAll, DWORD dwMilliseconds); – ожидает завершения одного или всех указанных объектов синхронизации (например, потоков).
- HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId); – создаёт новый поток выполнения в текущем процессе.
- BOOL CloseHandle(HANDLE hObject); – закрывает дескриптор объекта ядра (в том числе потока), освобождая связанные с ним ресурсы ОС.
- BOOL QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency); – получает частоту высокоточного счётчика производительности (в тактах в секунду). Используется для калибровки измерений времени.
- BOOL QueryPerformanceCounter(LARGE_INTEGER *lpPerformanceCount); – считывает текущее значение высокоточного счётчика производительности. Используется совместно с QueryPerformanceFrequency для точного измерения интервалов времени.

Параллельная сортировка реализована с использованием потоков Windows API. Основной процесс разбивает исходный массив на участки заданного 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 высокоточных счётчиков производительности. 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 <windows.h>
#include <time.h>


#define ARRAY_SIZE 100000
#define RUN_SIZE 10000 // для ARRAY_SIZE == RUN_SIZE смысла от потоков не будет


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(n1 * sizeof(int));
    int * R = (int *)malloc(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(L);
```

```
    free(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);
        }
    }
}

DWORD WINAPI thread_sort(LPVOID param) {
    ThreadArgs * args = (ThreadArgs *)param;
    insertion_sort(args->arr, args->left, args->right);
    free(args);
    return 0;
}

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];
}

double get_time_ms() {
    static LARGE_INTEGER freq;
    static int initialized = 0;
    LARGE_INTEGER counter;

    if (!initialized) {
        QueryPerformanceFrequency(&freq);
        initialized = 1;
    }

    QueryPerformanceCounter(&counter);
    return (double)counter.QuadPart * 1000.0 / (double)freq.QuadPart;
}

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);
    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;
        HANDLE * threads = (HANDLE *)malloc(max_threads * sizeof(HANDLE));

        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(sizeof(ThreadArgs));
            arg->arr = arr;
            arg->left = left;
            arg->right = right;

            threads[active] = CreateThread(NULL, 0, thread_sort, arg, 0, NULL);
            if (threads[active] == NULL) {
                fprintf(stderr, "Thread creation failed!\n");
                free(arg);
                for (int i = 0; i < active; i++) CloseHandle(threads[i]);
                free(threads);
                exit(1);
            }
        }

        WaitForMultipleObjects(active, threads, TRUE, INFINITE);
        for (int i = 0; i < active; i++) CloseHandle(threads[i]);
        free(threads);
    }

    merge_all(arr, n, RUN_SIZE);

    double end = get_time_ms();
    return end - start;
}




int main() {
    srand((unsigned)time(NULL));

    int * arr_original = (int *)malloc(ARRAY_SIZE * sizeof(int));
    int * arr_copy = (int *)malloc(ARRAY_SIZE * sizeof(int));
    generate_array(arr_original, ARRAY_SIZE);
```

```
    printf("Array size: %d\n", ARRAY_SIZE);



    // Последовательная версия
    copy_array(arr_original, arr_copy, ARRAY_SIZE);
    double t_seq = sequential_sort(arr_copy, ARRAY_SIZE);
    printf("Sequential: %.3f ms\n", t_seq);



    // Параллельная версия
    int threads_to_test[] = {1, 2, 4, 8, 16, 32, 64, 128}; // у меня 16 логических
процессоров
    int num_tests = sizeof(threads_to_test) / sizeof(threads_to_test[0]);

    printf("\nThreads | Time (ms) | Speedup | Efficiency\n");
    printf("-----------------------------------------\n");

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



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

Протокол работы программы

➤ `./program.exe`

```
Array size: 100000
Sequential: 233.029 ms

Threads | Time (ms) | Speedup | Efficiency
-----------------------------------------
      1 |   232.486 |   1.002 |     100.23%
      2 |   122.409 |   1.904 |      95.18%
      4 |    79.518 |   2.931 |      73.26%
      8 |    64.622 |   3.606 |      45.08%
     16 |    43.508 |   5.356 |      33.47%
     32 |    48.373 |   4.817 |      15.05%
     64 |    46.664 |   4.994 |       7.80%
    128 |    46.004 |   5.065 |       3.96%
```

Вывод

Программа демонстрирует ускорение при использовании многопоточности, однако эффективность резко снижается после определённого количества потоков. Наибольшее ускорение (~5.36×) достигается при 16 потоках, что совпадает с 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 95% до 45%). Однако при превышении числа логических процессоров (32 и более потоков) ускорение не только перестаёт расти, но и снижается, а эффективность падает до единиц процентов. 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	232.486	1.002	100.23%
2	122.409	1.904	95.18%
4	79.518	2.931	73.26%
8	64.622	3.606	45.08%
16	43.508	5.356	33.47%
32	48.373	4.817	15.05%
64	46.664	4.994	7.80%
128	46.004	5.065	3.96%

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

$$Speedup = \frac{T_{seq}}{T_{par}}$$

где T_{seq} – время выполнения последовательного алгоритма, а T_{par} – время выполнения параллельного варианта при заданном числе потоков. Эффективность оценивает, насколько рационально используются вычислительные ресурсы, и рассчитывается как

$$Efficiency = \frac{Speedup}{p} * 100\%$$

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.