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: Телепнева А.В.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 15.10.25

Москва, 2025

Постановка задачи

Вариант 2.

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы. 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.

Задание: Отсортировать массив целых чисел при помощи параллельного алгоритма быстрой сортировки

Общий метод и алгоритм решения

Использованные системные вызовы:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg); - создаёт новый поток, который начинает выполнение функции start_routine.
- int pthread_join(pthread_t thread, void **retval); - ожидает завершения указанного потока.
- int sem_init(sem_t *sem, int pshared, unsigned int value); - инициализирует семафор, используемый для ограничения количества потоков.
- int sem_trywait(sem_t *sem); - пытается атомарно уменьшить значение семафора без блокировки. Используется для проверки возможности создания нового потока.
- int sem_post(sem_t *sem); - увеличивает значение семафора и сигнализирует о завершении работы потока.
- int sem_destroy(sem_t *sem); - уничтожает семафор после завершения работы программы.
- int clock_gettime(clockid_t clk_id, struct timespec *tp); - используется для измерения времени выполнения алгоритмов с применением монотонных часов.

Описание работы программы:

После запуска программа принимает один аргумент командной строки — максимальное количество потоков, которые могут одновременно выполняться.

Далее происходит генерация массива целых чисел фиксированного размера. Создаются две копии массива: одна используется для последовательной сортировки, вторая — для параллельной.

Сначала производится замер времени выполнения параллельной версии быстрой сортировки.

Затем выполняется последовательная версия алгоритма, и также измеряется время её работы.

По завершении сортировки программа выводит:

- количество используемых потоков;
- время выполнения параллельной и последовательной версии;
- ускорение, вычисляемое по формуле
  S = Ts / Tp,

  где Ts — время последовательной версии, Tp — время параллельной версии;

- эффективность, вычисляемую по формуле
  E = S / p,

  где p — количество потоков;

- результат проверки корректности сортировки.

Код программы

quicksort.c

```
#define _POSIX_C_SOURCE 200809L


#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <time.h>

#include <stdbool.h>


#define ARRAY_SIZE 1000000


typedef struct {

    int *arr;

    int left;

    int right;

} ThreadArgs;


static sem_t thread_limiter;
```

```
static int partition(int *arr, int left, int right) {
    int pivot = arr[(left + right) / 2];
    int i = left;
    int j = right;

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;

        if (i <= j) {
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    }
    return i;
}

static void quicksort_seq(int *arr, int left, int right) {
    if (left >= right) return;

    int index = partition(arr, left, right);
    quicksort_seq(arr, left, index - 1);
    quicksort_seq(arr, index, right);
}

static void *quicksort_par_wrapper(void *arg);
```

```
static void quicksort_par(int *arr, int left, int right) {
    if (left >= right) return;

    int index = partition(arr, left, right);

    if (sem_trywait(&thread_limiter) == 0) {
        pthread_t thread;
        ThreadArgs *args = malloc(sizeof(ThreadArgs));

        args->arr = arr;
        args->left = left;
        args->right = index - 1;

        if (pthread_create(&thread, NULL, quicksort_par_wrapper, args) == 0) {
            quicksort_par(arr, index, right);
            pthread_join(thread, NULL);
        } else {
            sem_post(&thread_limiter);
            free(args);
            quicksort_seq(arr, left, right);
        }
    } else {
        quicksort_seq(arr, left, right);
    }
}

static void *quicksort_par_wrapper(void *arg) {
    ThreadArgs *args = (ThreadArgs *)arg;
    quicksort_par(args->arr, args->left, args->right);
    free(args);
```

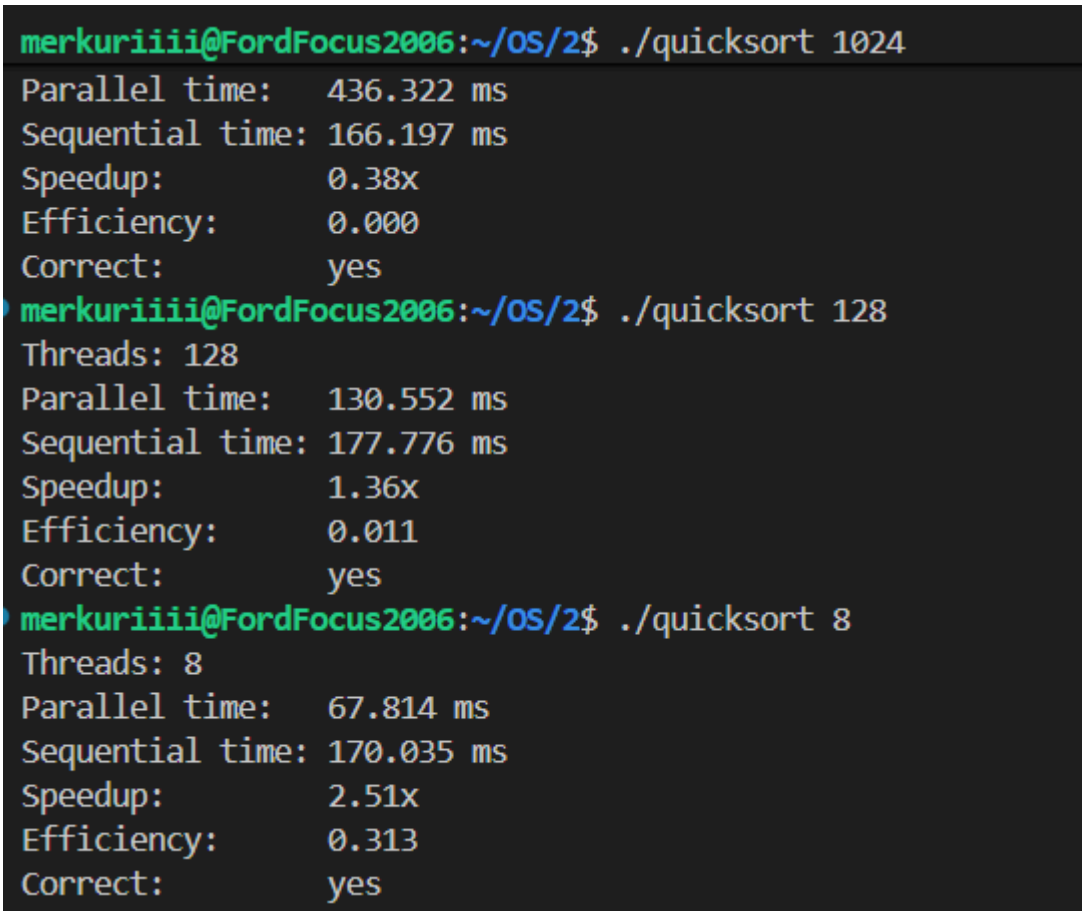
```
    sem_post(&thread_limiter);
    return NULL;
}

static void parallel_quicksort(int *arr, int n, int max_threads) {
    if (max_threads <= 1) {
        quicksort_seq(arr, 0, n - 1);
        return;
    }

    sem_init(&thread_limiter, 0, max_threads - 1);
    quicksort_par(arr, 0, n - 1);
    sem_destroy(&thread_limiter);
}

static bool is_sorted(int *arr, int n) {
    for (int i = 1; i < n; i++)
        if (arr[i] < arr[i - 1]) return false;
    return true;
}

int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <num_threads>\n", argv[0]);
        return 1;
    }

    int max_threads = atoi(argv[1]);
    if (max_threads < 1) max_threads = 1;
```

```
int *arr_par = malloc(sizeof(int) * ARRAY_SIZE);
int *arr_seq = malloc(sizeof(int) * ARRAY_SIZE);

if (!arr_par || !arr_seq) {
    perror("malloc");
    return 1;
}

srand((unsigned)time(NULL));
for (int i = 0; i < ARRAY_SIZE; i++) {
    arr_par[i] = rand();
    arr_seq[i] = arr_par[i];
}

struct timespec start, end;
// параллельн
clock_gettime(CLOCK_MONOTONIC, &start);
parallel_quicksort(arr_par, ARRAY_SIZE, max_threads);
clock_gettime(CLOCK_MONOTONIC, &end);

double par_time = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
// последовательн
clock_gettime(CLOCK_MONOTONIC, &start);
quicksort_seq(arr_seq, 0, ARRAY_SIZE - 1);
clock_gettime(CLOCK_MONOTONIC, &end);

double seq_time = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;

printf("Threads: %d\n", max_threads);
printf("Parallel time:   %.3f ms\n", par_time * 1000);
```

```
    printf("Sequential time: %.3f ms\n", seq_time * 1000);
    printf("Speedup:         %.2fx\n", seq_time / par_time);
    printf("Efficiency:      %.3f\n", (seq_time / par_time) / max_threads);
    printf("Correct:         %s\n", is_sorted(arr_par, ARRAY_SIZE) ? "yes" : "no");

    free(arr_par);
    free(arr_seq);
    return 0;
}
```

Протокол работы программы

Тестирование:



```
merkuriiii@FordFocus2006:~/OS/2$ ./quicksort 4
Threads: 4
Parallel time:   101.615 ms
Sequential time: 166.665 ms
Speedup:         1.64x
Efficiency:      0.410
Correct:         yes
merkuriiii@FordFocus2006:~/OS/2$ ./quicksort 16
Threads: 16
Parallel time:   161.241 ms
Sequential time: 167.573 ms
Speedup:         1.04x
Efficiency:      0.065
Correct:         yes
merkuriiii@FordFocus2006:~/OS/2$ ./quicksort 1
Threads: 1
Parallel time:   213.845 ms
Sequential time: 158.922 ms
Speedup:         0.74x
Efficiency:      0.743
Correct:         yes
```

```
merkuriiii@FordFocus2006:~/OS/2$ ./quicksort 2
Threads: 2
Parallel time:   105.781 ms
Sequential time: 155.343 ms
Speedup:         1.47x
Efficiency:      0.734
Correct:         yes
```

Анализ результатов

Число потоков	Время параллельное, мс	Время последовательное, мс	Ускорение	Эффективность
1	213.845	158.922	0.74	0.743
2	105.781	155.343	1.47	0.734
4	101.615	166.665	1.64	0.410
8	67.814	170.035	2.51	0.313
16	161.241	167.573	1.04	0.065

32	163.538	154.401	0.94	0.030
64	89.438	157.348	1.76	0.027
128	97.582	159.497	1.63	0.013
1024	436.322	166.197	0.38	0.000

Эксперимент показал, что ускорение параллельной версии быстрой сортировки достигает максимума при использовании 8 потоков (ускорение ≈ 2.51×). Это значение близко к типичному количеству логических ядер современных процессоров, что подтверждает теоретические ожидания.

При увеличении числа потоков свыше 8 наблюдается снижение ускорения и эффективности. Это связано с накладными расходами на создание и управление потоками, а также с конкуренцией за ресурсы (процессорное время, кэш, память). При очень большом количестве потоков (например, 1024) время выполнения значительно возрастает, а ускорение становится меньше единицы, что указывает на деградацию производительности из-за перегрузки планировщика ОС.

Таким образом, оптимальное количество потоков для данного алгоритма и аппаратной конфигурации составляет 8, что соответствует числу логических ядер системы. Дальнейшее увеличение потоков нецелесообразно, так как ведёт к снижению эффективности параллелизации.

Вывод

В ходе выполнения лабораторной работы была реализована программа, демонстрирующая использование многопоточности и средств синхронизации POSIX Threads на примере параллельного алгоритма быстрой сортировки.

Экспериментальные результаты показали, что при увеличении количества потоков время выполнения алгоритма уменьшается, а ускорение возрастает вплоть до значения, близкого к количеству логических ядер системы.
При дальнейшем увеличении числа потоков наблюдается снижение эффективности, что объясняется накладными расходами на создание и управление потоками, а также особенностями работы планировщика операционной системы и конкуренцией за вычислительные ресурсы.

Таким образом, можно сделать вывод, что оптимальное количество потоков для параллельного алгоритма соответствует количеству логических ядер процессора, а чрезмерное увеличение числа потоков не приводит к дополнительному ускорению и снижает эффективность параллельного исполнения.