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: Захарченко М. А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 30.10.25

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 (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.

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); – Создает поток с заданными атрибутами, который начинает выполнение функции start_routine
- int pthread_join(pthread_t thread, void **retval); – ожидает завершения указанного потока.
- int sem_trywait(sem_t *__sem); - атомарное уменьшение счетчика семафора без блокировки
- int sem_post(sem_t *__sem); -атомарное увеличение счетчика и пробуждение ожидающих потоков
- int sem_init(sem_t *__sem, int __pshared, unsigned int __value); - инициализация семафора; __sem - указатель на структуру семафора; __pshared - флаг разделения между процессами; __value - максиальное количество дополнительных потков
- int sem_destroy(sem_t *__sem) - уничтожение семафора;
- int clock_gettime(clockid_t clk_id, struct timespec *tp); – получает текущее монотоное время системы
  struct timespec {
  time_t tv_sec; - секунды
  long tv_nsec; - наносекунды
  };

В рамках лабораторной работы я реализовал алгоритм параллельной быстрой сортировки. В решении я использовал семафоры, ведь они идеально подходят для отслеживания количества созданных потоков, кроме этого, в программе нет данных, из-за которых могла бы произойти гонка процессов, а значит и мьютекс - не нужен. В программе реализовано два алгоритма - параллельный и последовательный. Также для параллельного алгоритма создана функция-обертка, для того, чтобы его можно было передавать как аргумент в функцию создания потока. Внутри самой функции происходит проверка, определяющая можно ли создать еще поток, или же передать все текущему потоку в последовательную версию алгоритма. Также была составлена таблица 1, в которой содержатся результаты работы программы, включающие в себя:

1. Число потоков
2. Время исполнения (мс)

3. Ускорение (рассчитывается по формуле: S = Ts / Tp, где Ts – время последовательной реализации, Tp – время параллельной реализации,1 <= S <= p, p – количество ядер)
4. Эффективность (рассчитывается по формуле: X = S / p, где X < 1)

Таблица 1

Число потоков	Время исполнения, мс	Ускорение	Эффективность
4	80.378	1.22	0.31
6	50.278	1.95	0.33
8	49.913	1.97	0.25
10	39.807	2.47	0.25
12	35.262	2.84	0.24
16	40.959	2.40	0.15
128	60.825	1.64	0.01
1024	137.011	0.61	0.001

Код программы

main.c

```c
#define _POSIX_C_SOURCE 200809L

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

#include <time.h>

#include <sys/time.h>

#include <stdbool.h>


static int num_threads = 1;

static sem_t thread_limiter;
```

```c
typedef struct {
    int *arr;
    size_t left;
    size_t right;
} ThreadArgs;


static size_t partition(int *arr, size_t left, size_t right) {
    int mid = arr[(left + right) / 2];
    int i = left, j = right - 1;
    int temp;
    do {
        while (i < right && arr[i] < mid) {
            ++i;
        }
        while (j >= left && arr[j] > mid) {
            --j;
        }
        if (i <= j) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            ++i;
            --j;
        }
    } while (i <= j);
    return j;
}


static void quicksort_seq(int *arr, size_t left, size_t right) {
```

```
    if (right - left < 2) {
        return;
    }
    size_t p = partition(arr, left, right);
    quicksort_seq(arr, left, p + 1);
    quicksort_seq(arr, p + 1, right);
}
static void *quicksort_par_wrapper(void *arg);
static void quicksort_par(int *arr, size_t left, size_t right) {
    if (right - left < 2) {
        return;
    }
    size_t p = partition(arr, left, right);
    if (sem_trywait(&thread_limiter) == 0) {
        ThreadArgs *thread_arg = malloc(sizeof(ThreadArgs));
        thread_arg->arr = arr;
        thread_arg->left = left;
        thread_arg->right = p + 1;
        pthread_t thread;
        if (pthread_create(&thread, NULL, quicksort_par_wrapper, thread_arg)
== 0) {
            quicksort_par(arr, p + 1, right);
            pthread_join(thread, NULL);
        } else {
            sem_post(&thread_limiter);
            free(thread_arg);
            quicksort_seq(arr, left, p + 1);
            quicksort_seq(arr, p + 1, right);
        }
    } else {
```

```
        quicksort_seq(arr, left, p + 1);
        quicksort_seq(arr, p + 1, right);
    }
}
static void *quicksort_par_wrapper(void *arg) {
    ThreadArgs *thread_arg = (ThreadArgs *)arg;
    quicksort_par(thread_arg->arr, thread_arg->left, thread_arg->right);
    free(thread_arg);
    sem_post(&thread_limiter);
    return NULL;
}
static void parallel_quicksort(int *arr, size_t n, int num_threads) {
    if (num_threads <= 1) {
        quicksort_seq(arr, 0, n);
        return;
    }

    if (sem_init(&thread_limiter, 0, num_threads - 1) != 0) {
        perror("sem_init");
        quicksort_seq(arr, 0, n);
        return;
    }
    quicksort_par(arr, 0, n);
    sem_destroy(&thread_limiter);
}

static bool is_sorted(int *arr, size_t n) {
    for (size_t 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;

    }


    num_threads = atoi(argv[1]);

    if (num_threads < 1) num_threads = 1;


    const size_t N = 1000000;


    int *arr_par = malloc(N * sizeof(int));

    if(arr_par == NULL){

        printf("MALLOC FAULT\n");

        return -1;

    }


    int *arr_seq = malloc(N * sizeof(int));

    if(arr_seq == NULL){

        free(arr_par);

        printf("MALLOC FAULT\n");

        return -1;

    }


    int num, index = 0;
```

```
    FILE* file = fopen("data_file.txt", "r");

    if(file == NULL){

        printf("Error: no such file!\n");

        free(arr_par);

        free(arr_seq);

        return -1;

    }



    while (fscanf(file, "%d", &num) == 1 && index < 1000000) {

        arr_par[index] = num;

        arr_seq[index] = num;

        ++index;

    }

    fclose(file);



    struct timespec start, end;



    clock_gettime(CLOCK_MONOTONIC, &start);

    parallel_quicksort(arr_par, N, num_threads);

    clock_gettime(CLOCK_MONOTONIC, &end);



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



    clock_gettime(CLOCK_MONOTONIC, &start);

    quicksort_seq(arr_seq, 0, N);

    clock_gettime(CLOCK_MONOTONIC, &end);



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

```c
    printf("Threads: %d\n", num_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("Efficienty:      %.3f\n", (seq_time / par_time) / num_threads);
    printf("Correct:         %s\n", is_sorted(arr_par, N) ? "yes" : "no");


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

Протокол работы программы

```
❯ ./main 4
Threads: 4
Parallel time:   80.378 ms
Sequential time: 98.101 ms
Speedup:         1.22x
Efficienty:      0.31
Correct:         yes
```

```
❯ ./main 6
Threads: 6
Parallel time:   50.278 ms
Sequential time: 98.274 ms
Speedup:         1.95x
Efficienty:      0.33
Correct:         yes
```

```
❯ ./main 8
Threads: 8
Parallel time:   49.913 ms
Sequential time: 98.127 ms
Speedup:         1.97x
Efficienty:      0.25
Correct:         yes
```

```
❯ ./main 10
Threads: 10
Parallel time:   39.807 ms
Sequential time: 98.403 ms
Speedup:         2.47x
Efficienty:      0.25
Correct:         yes
```

```
❯ ./main 12
Threads: 12
Parallel time:   35.262 ms
Sequential time: 100.131 ms
Speedup:         2.84x
Efficienty:      0.24
Correct:         yes
```

```
❯ ./main 16
Threads: 16
Parallel time:   40.959 ms
Sequential time: 98.314 ms
Speedup:         2.40x
Efficienty:      0.15
Correct:         yes
```

```
❯ ./main 128
Threads: 128
Parallel time:   60.825 ms
Sequential time: 99.967 ms
Speedup:         1.64x
Efficienty:      0.01
Correct:         yes
```

```
❯ ./main 1024
Threads: 1024
Parallel time:   126.789 ms
Sequential time: 99.692 ms
Speedup:         0.79x
Efficienty:      0.001
Correct:         yes
```

Вывод

В результате лабораторной работы была реализована программа, реализующая параллельный алгоритм согласно заданию, а именно Алгоритм быстрой параллельной сортировки. По таблице 1 видно, что ускорение увеличивается при увеличении количества потоков вплоть до значения в 12 потоков, после чего начинает падать. Благодаря этому, можно сделать вывод, что максимальное ускорение достигается, когда количество потоков равно количеству логических ядер на устройстве. Кроме этого, на ускорение и эффективность также может влиять состояние кэша и состояние планировщика ОС ведь не только наша программа потребялет ресурсы CPU, поэтому для наглядности выполненной работы, были выбраны лучшие результаты запусков программы.