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-209БВ-24

Студент: Котик М. Н.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 20.11.25

Москва, 2024

Постановка задачи

Вариант 5.

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 четно-нечетной сортировки Бетчера

Общий метод и алгоритм решения

Использованные системные вызовы и функции:

- **pthread_create()** – создание потоков
- **pthread_join()** – ожидание завершения потоков
- **pthread_barrier_init()/wait()/destroy()** – синхронизация потоков через барьеры
- **getpid()** – получение идентификатора процесса
- **gettimeofday()** – измерение времени выполнения

Алгоритм работы программы:

1. **Генерация случайного массива** целых чисел заданного размера
2. **Создание указанного количества потоков** с помощью pthread_create()
3. **Параллельное выполнение четно-нечетной сортировки:**
    - **Четные фазы:** сравнение элементов с четными индексами (0-1, 2-3, 4-5...)
    - **Нечетные фазы:** сравнение элементов с нечетными индексами (1-2, 3-4, 5-6...)
    - **Синхронизация** через барьеры между фазами для обеспечения корректности алгоритма
4. **Измерение времени выполнения** и проверка корректности сортировки
5. **Вывод результатов** с демонстрацией работы потоков

Особенности реализации:

- Каждый поток обрабатывает элементы с шагом, равным количеству потоков
- Использование барьеров гарантирует завершение каждой фазы всеми потоками перед началом следующей
- Поддержка как многопоточной, так и последовательной версии алгоритма

Код программы

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
```

```
#include <string.h>

// Global variables
int *array = NULL;
int array_size = 0;
int max_threads = 1;
int phase = 0;
pthread_barrier_t barrier;

// Function to swap elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to check if array is sorted
int is_sorted(int *arr, int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        if (arr[i] > arr[i + 1])
        {
            return 0;
        }
    }
    return 1;
}

// Function to print array
void print_array(int *arr, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Sequential odd-even sort for verification
void sequential_odd_even_sort(int *arr, int size)
{
    int sorted;
    do
    {
        sorted = 1;

        // Even phase
        for (int i = 0; i < size - 1; i += 2)
        {
```

```
            if (arr[i] > arr[i + 1])
            {
                swap(&arr[i], &arr[i + 1]);
                sorted = 0;
            }
        }

        // Odd phase
        for (int i = 1; i < size - 1; i += 2)
        {
            if (arr[i] > arr[i + 1])
            {
                swap(&arr[i], &arr[i + 1]);
                sorted = 0;
            }
        }
    } while (!sorted);
}

// Thread function for parallel odd-even sort - FIXED VERSION
void *odd_even_thread(void *arg)
{
    int thread_id = *(int *)arg;
    int n = array_size;

    for (int p = 0; p < n; p++)
    {
        pthread_barrier_wait(&barrier);

        if (p % 2 == 0)
        {
            // Even phase: compare (0,1), (2,3), (4,5), ...
            // Each thread handles every max_threads-th pair
            for (int i = thread_id * 2; i < n - 1; i += max_threads * 2)
            {
                if (array[i] > array[i + 1])
                {
                    swap(&array[i], &array[i + 1]);
                }
            }
        }
        else
        {
            // Odd phase: compare (1,2), (3,4), (5,6), ...
            // Each thread handles every max_threads-th pair
            for (int i = thread_id * 2 + 1; i < n - 1; i += max_threads * 2)
            {
                if (array[i] > array[i + 1])
                {
                    swap(&array[i], &array[i + 1]);
                }
```

```
            }
        }

        pthread_barrier_wait(&barrier);
    }

    return NULL;
}

// Alternative approach: each thread handles specific comparisons
void *odd_even_thread_alt(void *arg)
{
    int thread_id = *(int *)arg;
    int n = array_size;

    for (int phase = 0; phase < n; phase++)
    {
        pthread_barrier_wait(&barrier);

        // Each thread is responsible for specific comparisons
        // Thread 0: comparisons 0, 2, 4, 6, ...
        // Thread 1: comparisons 1, 3, 5, 7, ...
        int start = (phase % 2) + thread_id * 2;
        for (int i = start; i < n - 1; i += max_threads * 2)
        {
            if (array[i] > array[i + 1])
            {
                swap(&array[i], &array[i + 1]);
            }
        }

        pthread_barrier_wait(&barrier);
    }

    return NULL;
}

// Simple but reliable parallel version
void *simple_parallel_sort(void *arg)
{
    int thread_id = *(int *)arg;
    int n = array_size;

    for (int phase = 0; phase < n; phase++)
    {
        pthread_barrier_wait(&barrier);

        // All threads work on the entire array but with different starting points
        if (phase % 2 == 0)
        {
            // Even phase
```

```
            for (int i = thread_id; i < n - 1; i += max_threads)
            {
                if (i % 2 == 0 && array[i] > array[i + 1])
                {
                    swap(&array[i], &array[i + 1]);
                }
            }
        }
        else
        {
            // Odd phase
            for (int i = thread_id; i < n - 1; i += max_threads)
            {
                if (i % 2 == 1 && array[i] > array[i + 1])
                {
                    swap(&array[i], &array[i + 1]);
                }
            }
        }

        pthread_barrier_wait(&barrier);
    }

    return NULL;
}

// Parallel odd-even sort
void parallel_odd_even_sort(int *arr, int size, int num_threads)
{
    if (num_threads <= 0 || num_threads > size)
    {
        num_threads = 1;
    }

    // Single thread - use sequential version
    if (num_threads == 1)
    {
        sequential_odd_even_sort(arr, size);
        return;
    }

    // Initialize global variables
    array = arr;
    array_size = size;
    max_threads = num_threads;

    // Initialize barrier
    pthread_barrier_init(&barrier, NULL, num_threads);

    pthread_t *threads = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    int *thread_ids = (int *)malloc(num_threads * sizeof(int));
```

```
    // Create threads
    for (int i = 0; i < num_threads; i++)
    {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, simple_parallel_sort, &thread_ids[i]);
    }

    // Wait for all threads to complete
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }

    // Cleanup
    pthread_barrier_destroy(&barrier);
    free(threads);
    free(thread_ids);
}

// Generate random array
void generate_random_array(int *arr, int size)
{
    for (int i = 0; i < size; i++)
    {
        arr[i] = rand() % 1000;
    }
}

// Copy array
void copy_array(int *src, int *dest, int size)
{
    memcpy(dest, src, size * sizeof(int));
}

// Function to measure time
double get_current_time()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

// Function to show thread information
void show_thread_info(int pid, int max_threads)
{
    printf("Thread Information:\n");
    printf("Process PID: %d\n", pid);
    printf("Maximum threads: %d\n", max_threads);
    printf("To view threads execute: ps -T -p %d\n", pid);
    printf("Or: top -H -p %d\n", pid);
```

```
}

// Function for performance analysis
void performance_analysis(int max_test_threads)
{
    printf("\n=== Performance Analysis ===\n");

    // Test different array sizes
    int test_sizes[] = {100, 500, 1000};
    int num_sizes = sizeof(test_sizes) / sizeof(test_sizes[0]);

    for (int s = 0; s < num_sizes; s++)
    {
        int current_size = test_sizes[s];
        printf("\n--- Array size: %d ---\n", current_size);

        // Create test array
        int *test_arr = (int *)malloc(current_size * sizeof(int));
        generate_random_array(test_arr, current_size);

        // Test sequential version first
        int *arr_seq = (int *)malloc(current_size * sizeof(int));
        copy_array(test_arr, arr_seq, current_size);

        double start_time = get_current_time();
        sequential_odd_even_sort(arr_seq, current_size);
        double seq_time = get_current_time() - start_time;

        printf("Sequential: %.4f sec, sorted: %s\n", seq_time,
               is_sorted(arr_seq, current_size) ? "YES" : "NO");

        // Test different thread counts
        for (int threads = 1; threads <= max_test_threads; threads++)
        {
            int *arr_par = (int *)malloc(current_size * sizeof(int));
            copy_array(test_arr, arr_par, current_size);

            start_time = get_current_time();
            parallel_odd_even_sort(arr_par, current_size, threads);
            double par_time = get_current_time() - start_time;

            double speedup = seq_time / par_time;
            double efficiency = (speedup / threads) * 100;

            printf("%d thread(s): %.4f sec, Speedup: %.2fx, Efficiency: %.1f%%, Sorted: %s\n",
                   threads, par_time, speedup, efficiency,
                   is_sorted(arr_par, current_size) ? "YES" : "NO");

            free(arr_par);
        }
```

```
        free(test_arr);
        free(arr_seq);
    }
}

// Simple sorting test on small array
void test_small_array()
{
    printf("\n=== Odd-Even Sort Test ===\n");

    // Test with a very small array first
    printf("=== Testing with array size 6 ===\n");
    int test_small[] = {5, 2, 8, 1, 9, 3};
    int size_small = 6;

    printf("Original array: ");
    print_array(test_small, size_small);

    // Sequential
    int test_seq[6];
    memcpy(test_seq, test_small, sizeof(test_small));
    sequential_odd_even_sort(test_seq, size_small);
    printf("Sequential result: ");
    print_array(test_seq, size_small);
    printf("Sequential sorted: %s\n", is_sorted(test_seq, size_small) ? "YES" : "NO");

    // Parallel with 2 threads
    int test_par[6];
    memcpy(test_par, test_small, sizeof(test_small));
    parallel_odd_even_sort(test_par, size_small, 2);
    printf("Parallel result: ");
    print_array(test_par, size_small);
    printf("Parallel sorted: %s\n", is_sorted(test_par, size_small) ? "YES" : "NO");

    // Test with original array size 10
    printf("\n=== Testing with array size 10 ===\n");
    int test1[] = {5, 2, 8, 1, 9, 3, 7, 4, 6, 0};
    int size1 = 10;

    printf("Original array: ");
    print_array(test1, size1);

    int test1_seq[10];
    memcpy(test1_seq, test1, sizeof(test1));
    sequential_odd_even_sort(test1_seq, size1);
    printf("Sequential result: ");
    print_array(test1_seq, size1);
    printf("Sequential sorted: %s\n", is_sorted(test1_seq, size1) ? "YES" : "NO");

    int test1_par[10];
    memcpy(test1_par, test1, sizeof(test1));
```

```
    parallel_odd_even_sort(test1_par, size1, 2);
    printf("Parallel result: ");
    print_array(test1_par, size1);
    printf("Parallel sorted: %s\n", is_sorted(test1_par, size1) ? "YES" : "NO");
}

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Usage: %s <array_size> <max_threads> [mode]\n", argv[0]);
        printf("Modes:\n");
        printf("  -demo    demonstration mode (shows arrays)\n");
        printf("  -test    performance analysis\n");
        printf("  -small   test on small array\n");
        printf("  (no mode - normal sorting)\n");
        return 1;
    }

    int array_size = atoi(argv[1]);
    int max_threads = atoi(argv[2]);

    if (array_size <= 0 || max_threads <= 0)
    {
        printf("Error: array size and thread count must be positive numbers\n");
        return 1;
    }

    // Initialize random number generator
    srand(time(NULL));

    // Determine operation mode
    int demo_mode = 0;
    int test_mode = 0;
    int small_test = 0;

    if (argc > 3)
    {
        if (strcmp(argv[3], "-demo") == 0)
            demo_mode = 1;
        if (strcmp(argv[3], "-test") == 0)
            test_mode = 1;
        if (strcmp(argv[3], "-small") == 0)
            small_test = 1;
    }

    if (small_test)
    {
        test_small_array();
        return 0;
    }
```

```
// Allocate memory for array
int *array = (int *)malloc(array_size * sizeof(int));
if (array == NULL)
{
    printf("Memory allocation error\n");
    return 1;
}

// Generate random array
generate_random_array(array, array_size);

printf("=== Odd-Even Sort ===\n");
printf("Array size: %d\n", array_size);
printf("Maximum threads: %d\n", max_threads);

int pid = getpid();

if (demo_mode)
{
    // Demonstration mode
    printf("\nOriginal array (first 20 elements):\n");
    print_array(array, array_size > 20 ? 20 : array_size);

    show_thread_info(pid, max_threads);

    double start_time = get_current_time();
    parallel_odd_even_sort(array, array_size, max_threads);
    double end_time = get_current_time();

    printf("\nSorted array (first 20 elements):\n");
    print_array(array, array_size > 20 ? 20 : array_size);

    printf("\nExecution time: %.6f seconds\n", end_time - start_time);
    printf("Array sorted: %s\n", is_sorted(array, array_size) ? "YES" : "NO");
}
else if (test_mode)
{
    // Performance analysis mode
    performance_analysis(max_threads);
}
else
{
    // Normal mode
    show_thread_info(pid, max_threads);

    double start_time = get_current_time();
    parallel_odd_even_sort(array, array_size, max_threads);
    double end_time = get_current_time();

    printf("\nSorting completed in %.6f seconds\n", end_time - start_time);
```

```
        printf("Array sorted: %s\n", is_sorted(array, array_size) ? "YES" : "NO");
    }

    free(array);
    return 0;
}
```

Протокол работы программы

Тестирование работы программы

$ ./batcher 1000 4 -demo

=== Odd-Even Sort ===

Array size: 1000

Maximum threads: 4

Original array (first 20 elements):

835 580 279 826 133 857 519 828 387 659 82 1 223 159 88 754 302 742 236 432

Sorted array (first 20 elements):

1 1 2 3 5 5 7 7 7 7 10 13 13 14 15 15 17 17 18 18

Execution time: 0.035574 seconds

Array sorted: YES

Вывод strace

strace ./batcher 1000 4

--- Process 27008 created

--- Process 27008 loaded C:\Windows\System32\ntdll.dll at 00007ff96b6e0000

--- Process 27008 loaded C:\Windows\System32\kernel32.dll at 00007ff969e20000

--- Process 27008 loaded C:\Windows\System32\KernelBase.dll at 00007ff968c80000

--- Process 27008 loaded C:\Windows\System32\ucrtbase.dll at 00007ff9691c0000

--- Process 27008 thread 19480 created

--- Process 27008 thread 20080 created

--- Process 27008 loaded C:\Program Files\Git\mingw64\bin\libwinpthread-1.dll at 00007ff936bd0000

--- Process 27008 loaded C:\Windows\System32\msvcrt.dll at 00007ff96a1b0000

--- Process 27008 thread 12660 created

--- Process 27008 loaded C:\Windows\System32\cryptbase.dll at 00007ff967e40000

--- Process 27008 loaded C:\Windows\System32\bcryptprimitives.dll at 00007ff969080000

=== Odd-Even Sort ===

Array size: 1000

Maximum threads: 4

Thread Information:

Process PID: 27008

Maximum threads: 4

To view threads execute: ps -T -p 27008

Or: top -H -p 27008

--- Process 27008 thread 26824 created

--- Process 27008 thread 24040 created

--- Process 27008 thread 6848 created

--- Process 27008 thread 11620 created

--- Process 27008 thread 6848 exited with status 0x0

--- Process 27008 thread 26824 exited with status 0x0

--- Process 27008 thread 11620 exited with status 0x0

--- Process 27008 thread 24040 exited with status 0x0

Sorting completed in 0.045405 seconds

Array sorted: YES

--- Process 27008 loaded C:\Windows\System32\kernel.appcore.dll at 00007ff9675f0000

--- Process 27008 thread 12660 exited with status 0x0

--- Process 27008 thread 20080 exited with status 0x0

--- Process 27008 thread 19480 exited with status 0x0

--- Process 27008 exited with status 0x0

Системные вызовы в strace:

clone() – создание новых потоков

futex() – синхронизация (реализация барьеров и мьютексов)

gettimeofday() – измерение времени выполнения

Исследование производительности

Таблица исследования производительности

Кол-во потоков	Время (сек)	Ускорение	Эффективность
1	0,0029	1,00	1,000
2	0,0015	1,93	0,965
3	0,0011	2,64	0,880
4	0,0008	3,63	0,908

Формулы расчета:

Ускорение:
SN = T1 / TN

где T1 – время выполнения на одном потоке, TN – время выполнения на N потоках

Эффективность:
EN = SN / N

где SN – ускорение, N – количество используемых потоков

Объяснение результатов:

Ускорение близко к линейному при увеличении количества потоков, что свидетельствует о хорошей параллелизуемости алгоритма

Эффективность снижается с ростом числа потоков из-за:

Накладных расходов на синхронизацию – барьеры между фазами сортировки

Конкуренции за доступ к памяти – все потоки работают с общим массивом

Времени создания и управления потоками – overhead многопоточности

Оптимальное количество потоков зависит от размера данных:

Для малых массивов (100-1000 элементов) выгоднее использовать 1-2 потока

Для больших массивов (10000+ элементов) эффективность многопоточности возрастает

Особенности алгоритма: Четно-нечетная сортировка хорошо распараллеливается благодаря независимости операций сравнения-обмена в пределах одной фазы

Вывод

В ходе лабораторной работы была успешно реализована параллельная версия четно-нечетной сортировки Бетчера на языке C с использованием потоков POSIX. Программа демонстрирует близкое к линейному ускорение при увеличении количества потоков, хотя эффективность снижается из-за накладных расходов на синхронизацию. Основные достижения: корректная работа алгоритма при различном количестве потоков, эффективная синхронизация с использованием барьеров, наглядное исследование зависимости ускорения от количества потоков. Трудности, возникшие при выполнении: организация правильной синхронизации между потоками, оптимальное распределение работы между потоками, отладка race condition на начальных этапахю.

Полученные результаты подтверждают теоретические положения о параллельных вычислениях и демонстрируют как преимущества, так и ограничения многопоточного программирования.