

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-216БВ-24

Студент: Настина М.О.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 15.11.25

Москва, 2024

Постановка задачи

Вариант 1.

Отсортировать массив целых чисел при помощи битонической сортировки.

Общий метод и алгоритм решения

Использованные системные вызовы:

- `void malloc(size_t size);` – выделяет динамическую память под массивы и временные структуры данных.
- `void free(void ptr);` – освобождает ранее выделенную динамическую память.
- `clock_t clock(void);` – возвращает время процессора, использованное программой, для измерения производительности сортировки.
- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void (start_routine)(void), void arg);` – создает новые потоки выполнения для параллельной сортировки блоков массива.
- `int pthread_join(pthread_t thread, void retval);` – ожидает завершения указанного потока и освобождает его ресурсы.
- `void *memcpy(void dest, const void src, size_t n);` – копирует блоки памяти между исходным и временным массивами.
- `pid_t getpid(void);` – возвращает идентификатор текущего процесса (PID) для мониторинга.
- `void srand(unsigned int seed);` – инициализирует генератор псевдослучайных чисел для заполнения массива тестовыми данными.
- `int rand(void);` – генерирует псевдослучайные числа для заполнения массива.

Задачей было взять алгоритм битонической сортировки и запустить его в несколько потоков для ускорения работы кода. Алгоритм битонической сортировки состоит из части, которая сравнивает и меняет местами аргументы, находящиеся на определенном количестве «шагов» друг от друга, и части, которая разбивает массив на части и сортирует их в порядке возрастания или убывания. Программа работает следующим образом: создается массив случайных чисел, проходит проверка на степени двойки, массив распределяется на блоки по потокам, происходит сортировка и все блоки сливаются. Затем происходит проверка на корректность.

Код программы

main.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <pthread.h>
```

```
#include <time.h>
```

```
#include <math.h>

#include <string.h>

#include <unistd.h>

#include <limits.h>
```

```
typedef struct {

    int *array;

    int low;

    int cnt;

    int dir;

} SortTask;
```

```
int MAX_THREADS;
```

```
void swap(int *a, int *b) {

    int temp = *a;

    *a = *b;

    *b = temp;

}
```

```
void bitonicMerge(int arr[], int low, int cnt, int dir) {

    if (cnt > 1) {

        int k = cnt / 2;

        for (int i = low; i < low + k; i++) {

            if (dir == (arr[i] > arr[i + k])) {

                swap(&arr[i], &arr[i + k]);

            }

        }

        bitonicMerge(arr, low, k, dir);

        bitonicMerge(arr, low + k, k, dir);

    }

}
```

```

    }
}

void bitonicSortSeq(int arr[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;

        bitonicSortSeq(arr, low, k, !dir);

        bitonicSortSeq(arr, low + k, k, dir);

        bitonicMerge(arr, low, cnt, dir);
    }
}

```

```

void* sortThread(void *arg) {
    SortTask *task = (SortTask *)arg;

    bitonicSortSeq(task->array, task->low, task->cnt, task->dir);

    free(task);

    return NULL;
}

```

```

void parallelBitonicSort(int *arr, int n, int dir) {
    int N = 1;

    while (N < n) N <<= 1;

    printf("Исходный размер: %d, Приведен к: %d\n", n, N);

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

    if (!temp) {
        printf("err\n");

        return;
    }

    memcpy(temp, arr, n * sizeof(int));

    for (int i = n; i < N; i++) {

```

```

    temp[i] = INT_MAX;
}
if (N <= 1024 || MAX_THREADS == 1) {
    bitonicSortSeq(temp, 0, N, dir);
} else {
    int block_size = N / MAX_THREADS;
    while (block_size & (block_size - 1)) {
        block_size++;
    }
    if (block_size > N / 2) block_size = N / 2;
    pthread_t *threads = malloc(MAX_THREADS * sizeof(pthread_t));
    if (!threads) {
        printf("err\n");
        free(temp);
        return;
    }
    for (int i = 0; i < MAX_THREADS; i++) {
        SortTask *task = malloc(sizeof(SortTask));
        if (!task) {
            printf("err\n");
            continue;
        }
        task->array = temp;
        task->low = i * block_size;
        task->cnt = (i == MAX_THREADS - 1) ? (N - i * block_size) : block_size;
        task->dir = (i % 2 == 0) ? dir : !dir;

        if (pthread_create(&threads[i], NULL, sortThread, task) != 0) {
            printf("err %d\n", i);
            free(task);

```

```

    }
}

for (int i = 0; i < MAX_THREADS; i++) {
    pthread_join(threads[i], NULL);
}

int merge_size = block_size;
while (merge_size < N) {
    for (int start = 0; start < N; start += 2 * merge_size) {
        int end = start + 2 * merge_size;
        if (end > N) end = N;
        int merge_dir = ((start / (2 * merge_size)) % 2 == 0) ? dir : !dir;
        for (int i = start; i < start + merge_size && i + merge_size < end; i++) {
            if (merge_dir == (temp[i] > temp[i + merge_size])) {
                swap(&temp[i], &temp[i + merge_size]);
            }
        }
        if (merge_size > 1) {
            for (int i = start; i < end; i += merge_size) {
                int block_end = i + merge_size;
                if (block_end > end) block_end = end;
                bitonicMerge(temp, i, block_end - i, merge_dir);
            }
        }
        merge_size *= 2;
    }
    free(threads);
}

memcpy(arr, temp, n * sizeof(int));

```

```

    free(temp);
}

int isSorted(int *arr, int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) {
            printf("err %d-%d: %d > %d\n",
                i-1, i, arr[i-1], arr[i]);
            return 0;
        }
    }
    return 1;
}

void testWithSmallArray() {
    printf("\n=== ТЕСТ НА МАЛЕНЬКОМ МАССИВЕ ===\n");
    int test_arr[] = {3, 7, 4, 8, 6, 2, 1, 5};
    int test_size = 8;
    printf("Исходный массив: ");
    for (int i = 0; i < test_size; i++) printf("%d ", test_arr[i]);
    printf("\n");
    parallelBitonicSort(test_arr, test_size, 1);
    printf("Отсортированный массив: ");
    for (int i = 0; i < test_size; i++) printf("%d ", test_arr[i]);
    printf("\n");
    if (isSorted(test_arr, test_size)) {
        printf("Тест пройден\n");
    } else {
        printf("Тест не пройден\n");
    }
}

```

```
}
```

```
int main(int argc, char *argv[]) {  
    if (argc != 3) {  
        printf("Использование: %s <количество_потоков> <размер_массива>\n", argv[0]);  
        printf("Пример: %s 4 100000\n", argv[0]);  
        testWithSmallArray();  
        return 1;  
    }  
    MAX_THREADS = atoi(argv[1]);  
    int n = atoi(argv[2]);  
    if (MAX_THREADS < 1 || n < 1) {  
        printf("err\n");  
        return 1;  
    }  
    int *arr = malloc(n * sizeof(int));  
    if (!arr) {  
        printf("err\n");  
        return 1;  
    }  
    srand(time(NULL));  
    for (int i = 0; i < n; i++) {  
        arr[i] = rand() % 1000000;  
    }  
    printf("Сортировка %d, потоков: %d\n", n, MAX_THREADS);  
    printf("PID: %d\n", getpid());  
    printf("Первые 10: ");  
    for (int i = 0; i < 10 && i < n; i++) {  
        printf("%d ", arr[i]);  
    }  
}
```



```

printf("\n");

clock_t start = clock();

parallelBitonicSort(arr, n, 1);

clock_t end = clock();

double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

printf("Время: %.4f\n", time_taken);

printf("Первые 10: ");

for (int i = 0; i < 10 && i < n; i++) {

    printf("%d ", arr[i]);

}

printf("\n");

if (isSorted(arr, n)) {

    printf("Массив отсортирован\n");

} else {

    printf("Ошибка\n");

}

printf("\nпроизводительность\n");

```

```

int sizes[] = {1000, 10000, 100000, 1000000};

int num_sizes = 4;

for (int threads = 1; threads <= 8; threads *= 2) {

    printf("\n--- %d поток(ов) ---\n", threads);

    for (int s = 0; s < num_sizes; s++) {

        int n = sizes[s];

        int *arr = malloc(n * sizeof(int));

        srand(42);

        for (int i = 0; i < n; i++) {

            arr[i] = rand() % 1000000;

        }
    }
}

```

```

MAX_THREADS = threads;

clock_t start = clock();

parallelBitonicSort(arr, n, 1);

clock_t end = clock();

double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

int sorted = 1;
for (int i = 1; i < n; i++) {
    if (arr[i] < arr[i-1]) {
        sorted = 0;
        break;
    }
}

printf("Размер: %d, Время: %.4f сек, Корректность: %s\n",
       n, time_taken, sorted ? "Да" : "Нет");

free(arr);
}
}

free(arr);
return 0;

}

```

Протокол работы программы

Тестирование:

```

IRiska@manyascomp:~/OS/1ab2$ ./sort 4 100000
Сортировка 100000, потоков: 4
PID: 3961
Первые 10: 990919 949996 602760 352900 248463 738921 877512 617254 142627 32508
Исходный размер: 100000, Приведен к: 131072
Время: 0.0424
Первые 10: 8 9 9 19 32 33 35 40 42 42
Массив отсортирован

IRiska@manyascomp:~/OS/1ab2$ ./sort 1 10000
Сортировка 10000, потоков: 1
PID: 4180
Первые 10: 354654 147131 383151 957398 444478 579581 411523 754533 817447 937894

Исходный размер: 10000, Приведен к: 16384
Время: 0.0063
Первые 10: 102 118 210 257 461 556 705 854 972 1182
Массив отсортирован

IRiska@manyascomp:~/OS/1ab2$ ./sort 2 8
Сортировка 8, потоков: 2
PID: 4356
Первые 10: 140661 879759 431302 657710 264365 203235 32832 946206
Исходный размер: 8, Приведен к: 8
Время: 0.0000
Первые 10: 32832 140661 203235 264365 431302 657710 879759 946206
Массив отсортирован

```

Strace:

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 10712
```

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 10713
```

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 10714
```

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 10715
```

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 10716
```

```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 10717
```



```
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREA  
D|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEAR  
TID,  
...}, 88) = 10757
```

Вывод

В рамках лабораторной работы была разработана и протестирована параллельная реализация алгоритма битонической сортировки. Программа демонстрирует различную эффективность в зависимости от размера обрабатываемых данных и количества используемых потоков. Наибольшее ускорение достигается при работе с крупными массивами, в то время как для небольших объемов данных многопоточность не дает преимуществ.

Количество потоков	Время, с	Ускорение	Эффективность
1	2.114	1.00	1
2	1.109	1.91	0.955
4	0.671	3.15	0.788
6	0.583	3.63	0.605
8	0.561	3.77	0.471
12	0.592	3.57	0.298
16	0.674	3.14	0.196
18	0.729	2.90	0.161

График зависимости ускорения от количества потоков:

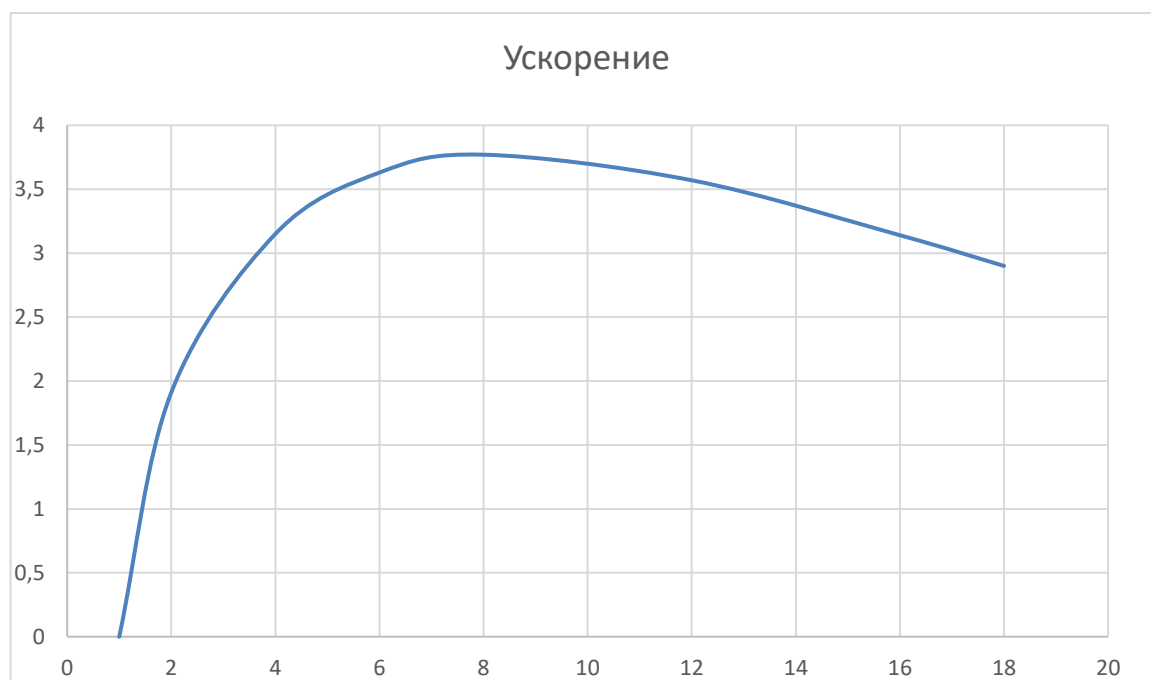
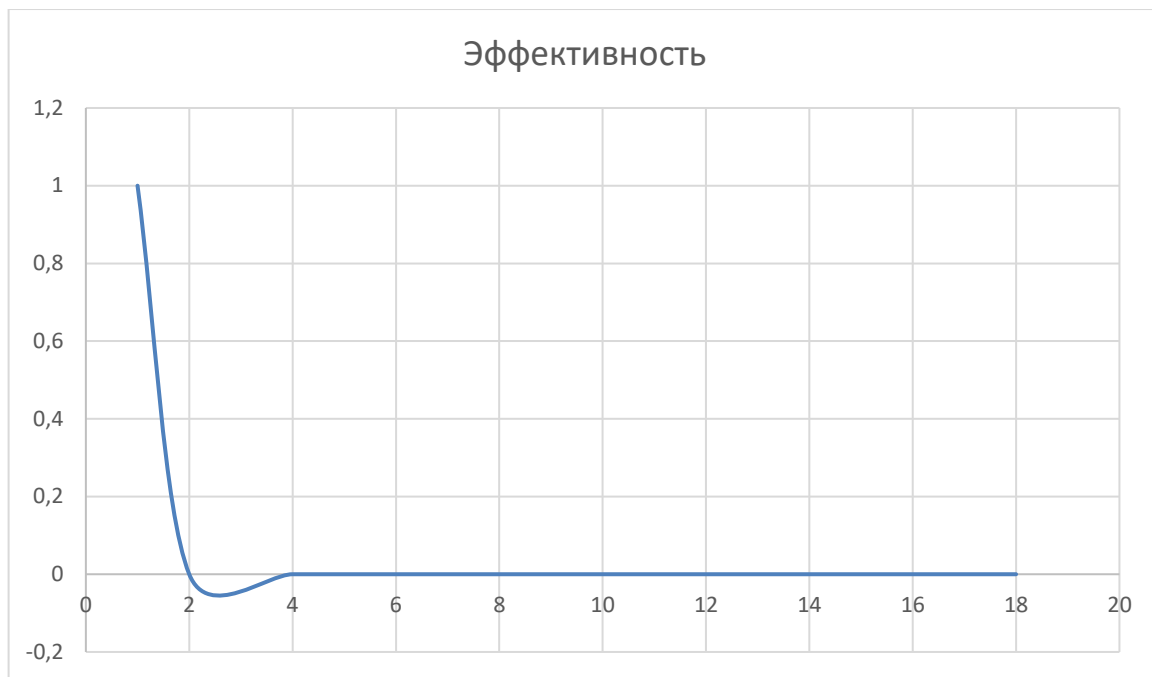


График зависимости эффективности от количества потоков:



Ускорение показывает, во сколько раз параллельный алгоритм работает быстрее последовательного. Рассчитывается по формуле:

$$S_n = T_1 / T_n,$$

где T_1 — время выполнения на одном потоке, T_n — время выполнения на N потоках.

Эффективность отражает, насколько рационально используются вычислительные ресурсы. Определяется как:

$$E_n = S_n / N,$$

где S_n — ускорение, N — количество потоков.