

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-210Б-23  
Студент: Григорян А.А.  
Преподаватель: Бахарев В.Д.  
Оценка: \_\_\_\_\_  
Дата: 20.02.25

Москва, 2025

## Постановка задачи

**Цель работы:**

**Целью является приобретение практических навыков в:**

- Управление потоками в ОС
- Обеспечение синхронизации между

**потоками Задание:**

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.

**Вариант 1) Отсортировать массив целых чисел при помощи битонической сортировки**

## Общий метод и алгоритм решения

**Использованные системные вызовы:**

- `void exit(int __status);` – выполняет немедленное завершение программы. Все используемые программой потоки закрываются, и временные файлы удаляются, управление возвращается ОС или другой программе.
- `int pthread_create(pthread_t *__restrict __newthread, const pthread_attr_t *__restrict __attr, void *(*__start_routine)(void *), void *__restrict __arg)` — создаёт поток с рутиной (стартовой функцией) и заданными аргументами
- `int pthread_join(pthread_t __th, void **_thread_return)` — дожидается завершения потока.
- `pthread_exit()` - Функция `pthread_exit()` завершает вызывающий поток и возвращает значение через `retval`, которое (если поток можно объединить) доступно другому потоку в том же процессе, который вызывает `pthread_join(3)`.

Программа получает на вход один аргумент - максимальное количество потоков

После создаётся нужное количество потоков, которые хранятся в массиве `threads`. Также, для каждого потока существует структура, которая указывает потоку на подмассив, который ему необходимо отсортировать. Структуры хранятся в массиве `sort_args`.

Структура `SortArgs`:

- **arr** — указатель на массив, который нужно отсортировать.
- **low** — индекс массива, с которого потоку необходимо начать сортировку.
- **cnt** — количество элементов в подмассиве, который сортирует поток.
- **dir** — направление сортировки. 1 - возрастание, 0 - убывание
- **thread\_id** — id потока.

После считывания числа потоков, программа создает динамический массив с длиной  $10^6$ , и заполняет его случайными целыми числами.

Потоки создаются в цикле, в этом же цикле заполняются поля структур.

Алгоритм битонической сортировки реализован в функциях `bitonic_sort` и `bitonic_merge`

Обе эти функции принимают следующие параметры:

- 1) **arr** - указатель на массив, который надо отсортировать
- 2) **low** - индекс, с которого начинается сортировка массива
- 3) **cnt** - количество элементов массива
- 4) **dir** — направление сортировки. 1 - возрастание, 0 - убывание

`Bitonic_sort` рекурсивно делит массив на два подмассива, если количество элементов больше одного. Один из подмассивов должен быть упорядочен по возрастанию, по этому передается `dir == 1`, другой - по убыванию, соответственно, `dir == 0`. После деления на подмассивы, вызывается функция `Bitonic_merge`, которая сливает два подмассива в один, меняя местами элементы попарно, индексы которых отличаются на  $k$  - половина количества элементов изначального массива. После первой попытки слияния, два подмассива являются отсортированными относительно друг друга (сравниваются элементы, индексы которых отличаются на  $k$ ). Может возникнуть случай, когда оба или один подмассив не отсортирован относительно самого себя. Для того, чтобы обработать этот случай, рекурсивно дважды вызывается функция `Bitonic_merge`, для сортировки обоих подмассивов относительно себя.

Для реализации многопоточности, была написана функция `Bitonic_sort_thread`, которая запускает работу `Bitonic_sort` в разных потоках, на основе структур этих потоков. Во время создания потоков с помощью функции `pthread_create`, передается адрес функции `Bitonic_sort_thread` и структура потока в качестве аргумента этой функции.

Для того чтобы разграничить массив для разных потоков, используется переменная `chunk_size`, на основе которой заполняются поля структуры `low` и `cnt`. Таким образом, массив равномерно разграничивается для разных потоков, отчего потоки не могут обращаться к данным, к которым обращаются другие потоки, состояние гонки данных не наступает.

После того, как потоки отсортировали свои подмассивы и завершились (основной поток дождался их завершения с помощью `pthread_join`), основному потоку необходимо слить все подмассивы в один (т.е. отсортировать подмассивы относительно их самих). Для этого основным потоком вызывается функция `Bitonic_merge`.

| Число потоков | Время исполнения (мс) | Ускорение | Эффективность |
|---------------|-----------------------|-----------|---------------|
| 1             | 312.377               | 1         | 1             |
| 2             | 162.673               | 1,92      | 0,96          |
| 3             | 110.759               | 2,82      | 0,94          |
| 4             | 95.703                | 3,26      | 0,815         |
| 5             | 91.703                | 3,4       | 0,68          |
| 6             | 84.689                | 3,68      | 0,613         |
| 7             | 79.231                | 3,94      | 0,56          |
| 16            | 49.671                | 6,28      | 0,39          |

## Код программы

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

```
typedef struct {
    int *arr;
    int low;
    int cnt;
    int dir;
    int thread_id;
} SortArgs;
```

```
void bitonic_merge(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])) {
                int temp = arr[i];
                arr[i] = arr[i + k];
                arr[i + k] = temp;
            }
        }
        bitonic_merge(arr, low, k, dir);
        bitonic_merge(arr, low + k, k, dir);
    }
}
```

```
void bitonic_sort(int arr[], int low, int cnt, int dir) {
    if (cnt > 1) {
        int k = cnt / 2;
        bitonic_sort(arr, low, k, 1);
        bitonic_sort(arr, low + k, k, 0);
    }
}
```

```

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

void* bitonic_sort_thread(void *args) {
    SortArgs *sort_args = (SortArgs*)args;
    //printf("Thread %d started\n", sort_args->thread_id); // Выводим информацию о запуске потока

    bitonic_sort(sort_args->arr, sort_args->low, sort_args->cnt, sort_args->dir);

    //printf("Thread %d finished\n", sort_args->thread_id); // Выводим информацию о завершении потока
    return NULL;
}

// Функция для записи массива в файл
void write_to_file(int arr[], int n, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (!file) {
        perror("Error opening file");
        exit(1);
    }

    fprintf(file, "Result: ");
    for (int i = 0; i < n; i++) {
        fprintf(file, "%d ", arr[i]);
    }

    fclose(file);
}

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

    int num_threads = atoi(argv[1]);
    if (num_threads <= 0) {
        fprintf(stderr, "Invalid number of threads\n");
        return 1;
    }

    int n = 20;
    int *arr = (int*)malloc(n * sizeof(int));

    if (arr == NULL) {
        perror("Memory allocation failed");
        return 1;
    }

    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;
    }
}

```

```

}

struct timeval start_time, end_time;
gettimeofday(&start_time, NULL);

pthread_t *threads = (pthread_t*)malloc(num_threads * sizeof(pthread_t));
SortArgs *sort_args = (SortArgs*)malloc(num_threads * sizeof(SortArgs));

int chunk_size = n / num_threads;

for (int i = 0; i < num_threads; i++) {
    sort_args[i].arr = arr;
    sort_args[i].low = i * chunk_size;
    sort_args[i].cnt = (i == num_threads - 1) ? n - (i * chunk_size) : chunk_size;
    sort_args[i].dir = 1;
    sort_args[i].thread_id = i;

    pthread_create(&threads[i], NULL, bitonic_sort_thread, &sort_args[i]);
}

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

bitonic_merge(arr, 0, n, 1);
gettimeofday(&end_time, NULL);

double time_taken = (end_time.tv_sec - start_time.tv_sec) +
    (end_time.tv_usec - start_time.tv_usec) / 1000000.0;

write_to_file(arr, n, "output.txt");

printf("Time taken for sorting: %.6f seconds\n", time_taken);

free(arr);
free(threads);
free(sort_args);

return 0;
}

```

## Протокол

```
[arcsenius@ars-nbdewxxx9 src]$ strace ./a.out 8
execve("./a.out", ["/a.out", "8"], 0x7ffe5acb0be8 /* 66 vars */) = 0
brk(NULL) = 0x56f1fe71c000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (Нет такого файла или каталога)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=241615, ...}) = 0
mmap(NULL, 241615, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7b6d55ba8000
close(3) = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\340_\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2014520, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7b6d55ba6000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0@\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2034616, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7b6d559b5000
mmap(0x7b6d559d9000, 1511424, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x7b6d559d9000
mmap(0x7b6d55b4a000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x195000) = 0x7b6d55b4a000
mmap(0x7b6d55b98000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e3000) = 0x7b6d55b98000
mmap(0x7b6d55b9e000, 31672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7b6d55b9e000
close(3) = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7b6d559b2000
arch_prctl(ARCH_SET_FS, 0x7b6d559b2740) = 0
set_tid_address(0x7b6d559b2a10) = 5589
set_robust_list(0x7b6d559b2a20, 24) = 0
rseq(0x7b6d559b3060, 0x20, 0, 0x53053053) = 0
mprotect(0x7b6d55b98000, 16384, PROT_READ) = 0
mprotect(0x56f1e64c7000, 4096, PROT_READ) = 0
mprotect(0x7b6d55c1d000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7b6d55ba8000, 241615) = 0
getrandom("\xa4\x8e\xf9\x88\xab\x32\x2e\xe7", 8, GRND_NONBLOCK) = 8
brk(NULL) = 0x56f1fe71c000
brk(0x56f1fe73d000) = 0x56f1fe73d000
rt_sigaction(SIGRT_1, {sa_handler=0x7b6d55a462b0, sa_mask=[], sa_flags=SA_RESTORER|SA_ONSTACK|SA_RESTART|SA_SIGINFO, sa_restorer=0x7b6d559f21d0}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7b6d551b1000
mprotect(0x7b6d551b2000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, child_tid=0x7b6d559b1990, parent_tid=0x7b6d559b1990, exit_signal=0, stack=0x7b6d551b1000, stack_size=0x7fff80, tls=0x7b6d559b16c0} => {parent_tid=[5590]}, 88) = 5590
```

```
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d549b0000
mprotect(0x7b6d549b1000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({ flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
VSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d551b0990, paren
t_tid=0x7b6d551b0990, exit_signal=0, stack=0x7b6d549b0000, stack_size=0x7fff80, tls=0x7b6d551b06c0} =>
{parent_tid=[5591]}, 88) = 5591
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d541af000
mprotect(0x7b6d541b0000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({ flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
VSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d549af990, paren
t_tid=0x7b6d549af990, exit_signal=0, stack=0x7b6d541af000, stack_size=0x7fff80, tls=0x7b6d549af6c0} =>
{parent_tid=[5592]}, 88) = 5592
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d539ae000
mprotect(0x7b6d539af000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({ flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
VSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d541ae990, paren
t_tid=0x7b6d541ae990, exit_signal=0, stack=0x7b6d539ae000, stack_size=0x7fff80, tls=0x7b6d541ae6c0} =>
{parent_tid=[5593]}, 88) = 5593
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d531ad000
mprotect(0x7b6d531ae000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({ flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
VSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d539ad990, paren
t_tid=0x7b6d539ad990, exit_signal=0, stack=0x7b6d531ad000, stack_size=0x7fff80, tls=0x7b6d539ad6c0} =>
{parent_tid=[5594]}, 88) = 5594
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d529ac000
mprotect(0x7b6d529ad000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({ flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
VSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d531ac990, paren
t_tid=0x7b6d531ac990, exit_signal=0, stack=0x7b6d529ac000, stack_size=0x7fff80, tls=0x7b6d531ac6c0} =>
{parent_tid=[5595]}, 88) = 5595
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d521ab000
mprotect(0x7b6d521ac000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({ flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
```



```

VSEM|CLONE_SETTSL|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d529ab990, paren
t_tid=0x7b6d529ab990, exit_signal=0, stack=0x7b6d521ab000, stack_size=0x7fff80, tls=0x7b6d529ab6c0} =>
{parent_tid=[5596]}, 88) = 5596
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) =
0x7b6d519aa000
mprotect(0x7b6d519ab000, 8388608, PROT_READ|PROT_WRITE) = 0
rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYS
VSEM|CLONE_SETTSL|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
child_tid=0x7b6d521aa990, paren
t_tid=0x7b6d521aa990, exit_signal=0, stack=0x7b6d519aa000, stack_size=0x7fff80, tls=0x7b6d521aa6c0} =>
{parent_tid=[5597]}, 88) = 5597
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
munmap(0x7b6d551b1000, 8392704) = 0
munmap(0x7b6d549b0000, 8392704) = 0
munmap(0x7b6d541af000, 8392704) = 0
munmap(0x7b6d539ae000, 8392704) = 0
openat(AT_FDCWD, "output.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
write(3, "Result: 27 101 148 296 189 307 7"..., 87) = 87
close(3) = 0
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0x4), ...}) = 0
write(1, "Time taken for sorting: 0.001270"..., 41Time taken for sorting: 0.001270 seconds
) = 41
exit_group(0) = ?
+++ exited with 0 +++

```

## Вывод

В ходе написания данной лабораторной работы я научился создавать программы, работающие с несколькими потоками, а также синхронизировать их между собой. В результате тестирования программы, я проанализировал каким образом количество потоков влияет на эффективность и ускорение работы программы. Оказалось, что большое количество потоков даёт хорошее ускорение на больших количествах входных данных, но эффективность использования ресурсов находится на приемлемом уровне только на небольшом количестве потоков, не превышающем количества логических ядер процессора. Лабораторная работа была довольно интересна, так как я впервые работал с многопоточностью и синхронизацией на СИ. Самой сложной подзадачей мне показалась написание алгоритма сортировки