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 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.**

**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. **сnt** - 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 к - половина количества элементов изначального массива.После первой попытки слияния, два подмассива являются отсортированнами относительно друг друга (сравниваются элементы, индексы которых отличаются на к). 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 разграничиваются 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 слиять 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-nbdewxx9 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\0\3\0>\0\1\0\0\0\340\_\2\0\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\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\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, paren

t\_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\_SYSVSEM|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\_SYSVSEM|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\_SYSVSEM|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\_SYSVSEM|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\_SYSVSEM|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\_SYSVSEM|CLONE\_SETTLS|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\_SYSVSEM|CLONE\_SETTLS|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