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-215Б-23

Студент: Кармишен Е.С.

Преподаватель: Миронов Е.С. (ПМИ)

Оценка: _____

Дата: 14.02.24

Москва, 2024

Постановка задачи

Вариант 4.

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.

Отсортировать массив целых чисел при помощи TimSort

Общий метод и алгоритм решения

Использованные системные вызовы:

- **malloc** — выделяет память в куче для динамических массивов и структур.
- **free** — освобождает память, ранее выделенную с помощью malloc.
- **pthread_create** — создает новый поток и запускает выполнение функции в этом потоке.
- **pthread_join** — ожидает завершения выполнения указанного потока.
- **pthread_exit** — завершает выполнение текущего потока.
- **gettimeofday** — получает текущее время с точностью до микросекунд.
- **srand** — инициализирует генератор случайных чисел.
- **rand** — генерирует псевдослучайное число.
- **time** — возвращает текущее время в секундах (используется для инициализации генератора случайных чисел).
- **atoi** — преобразует строку в целое число (используется для преобразования аргумента командной строки в число потоков).

Алгоритм работы программы:

1. Инициализация и ввод данных:
   - Программа начинает с обработки аргументов командной строки. Она ожидает один аргумент — максимальное количество потоков, которые будут использоваться для сортировки.
   - Затем программа запрашивает у пользователя размер массива, который нужно отсортировать.
   - После этого программа выделяет память для массива с помощью функции `malloc`.
   - Далее программа инициализирует генератор случайных чисел с помощью функции `srand` и заполняет массив случайными числами с помощью функции `rand`.
2. Вывод исходного массива:
   - Программа выводит исходный массив на экран для пользователя.
3. Измерение времени выполнения:
   - Перед началом сортировки программа записывает текущее время с помощью функции `gettimeofday`.

4. Многопоточная сортировка:
    - Программа вызывает функцию `timSort`, которая выполняет сортировку массива с использованием алгоритма TimSort.
    - Внутри функции `timSort` программа создает массив структур `ThreadData`, каждая из которых содержит часть массива для сортировки, и массив дескрипторов потоков `pthread_t`.
    - Программа рассчитывает размер части массива, которую будет сортировать каждый поток, и создает потоки с помощью функции `pthread_create`. Каждый поток выполняет функцию `timSortThread`, которая сортирует свою часть массива.
    - После создания всех потоков программа ожидает их завершения с помощью функции `pthread_join`.
5. Сортировка внутри потоков:
    - Внутри функции `timSortThread` программа выполняет сортировку вставками (`insertionSort`) для небольших частей массива, а затем объединяет отсортированные части с помощью функции `merge`.
    - Функция `insertionSort` выполняет сортировку вставками для заданного диапазона массива. Этот метод эффективен для небольших массивов и подмассивов, что делает его подходящим для начальной стадии сортировки.
    - Функция `merge` объединяет два отсортированных подмассива в один отсортированный массив. Этот процесс повторяется, пока весь массив не будет отсортирован.
6. Завершение сортировки и измерение времени:
    - После завершения сортировки программа снова записывает текущее время с помощью функции `gettimeofday` и вычисляет время, затраченное на сортировку.
7. Вывод отсортированного массива и статистики:
    - Программа выводит отсортированный массив на экран.
    - Затем программа выводит время, затраченное на сортировку, и количество использованных потоков.
8. Освобождение памяти:
    - В конце программа освобождает память, выделенную для массива и структур данных, с помощью функции `free`.

Код программы

timsort2.c

```c
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <string.h>

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

```c
#include <time.h>


#define MIN_MERGE 32

#define MIN(a, b) ((a) < (b) ? (a) : (b))


// структура передачи данных в поток

typedef struct {

int *array;

int left;

int right;

} ThreadData;


// сортировка вставками

void insertionSort(int arr[], int left, int right) {

for (int i = left + 1; i <= right; i++) {

int temp = arr[i];

int j = i - 1;

while (j >= left && arr[j] > temp) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = temp;

}

}


//слияние двух отсортированных подмассивов

void merge(int arr[], int l, int m, int r) {

int len1 = m - l + 1, len2 = r - m;

int *left = (int *)malloc(len1 * sizeof(int));

int *right = (int *)malloc(len2 * sizeof(int));
```

```
for (int i = 0; i < len1; i++)

left[i] = arr[l + i];

for (int i = 0; i < len2; i++)

right[i] = arr[m + 1 + i];



int i = 0, j = 0, k = l;

while (i < len1 && j < len2) {

if (left[i] <= right[j])

arr[k++] = left[i++];

else

arr[k++] = right[j++];

}



while (i < len1)

arr[k++] = left[i++];



while (j < len2)

arr[k++] = right[j++];



free(left);

free(right);

}



//сортировка timsort, выполняемая в потоке

void *timSortThread(void *arg) {

ThreadData *data = (ThreadData *)arg;

int *arr = data->array;

int left = data->left;

int right = data->right;



int minRun = MIN_MERGE;
```

```
for (int start = left; start <= right; start += minRun) {
int end = MIN(start + minRun - 1, right);
insertionSort(arr, start, end);
}

for (int size = minRun; size < right - left + 1; size = 2 * size) {
for (int start = left; start <= right; start += 2 * size) {
int mid = start + size - 1;
int end = MIN(start + 2 * size - 1, right);
if (mid < end) {
merge(arr, start, mid, end);
}
}
}

pthread_exit(NULL);
}

//основная ф-я timsort
void timSort(int arr[], int n, int maxThreads) {
if (n < 2) return;

pthread_t *threads = (pthread_t *)malloc(maxThreads * sizeof(pthread_t));
ThreadData *threadData = (ThreadData *)malloc(maxThreads * sizeof(ThreadData));

int chunkSize = (n + maxThreads - 1) / maxThreads;
for (int i = 0; i < maxThreads; i++) {
int start = i * chunkSize;
int end = MIN(start + chunkSize - 1, n - 1);
if (start >= n) break;
```

```
threadData[i].array = arr;

threadData[i].left = start;

threadData[i].right = end;



pthread_create(&threads[i], NULL, timSortThread, (void *)&threadData[i]);

}



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

pthread_join(threads[i], NULL);

}



// Слияние частей массива

for (int size = chunkSize; size < n; size = 2 * size) {

for (int left = 0; left < n; left += 2 * size) {

int mid = left + size - 1;

int right = MIN(left + 2 * size - 1, n - 1);

if (mid < right) {

merge(arr, left, mid, right);

}

}

}



free(threads);

free(threadData);

}



int main(int argc, char *argv[]) {

if (argc < 2) {

fprintf(stderr, "Usage: %s <max_threads>\n", argv[0]);

return 1;

}
```

```
int maxThreads = atoi(argv[1]); // Преобразование аргумента в число (количество потоков)

int arraySize;


printf("Enter the number of elements in the array: ");

scanf("%d", &arraySize);


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


// Инициализация генератора случайных чисел

srand(time(NULL));


// Генерация случайных чисел для массива

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

arr[i] = rand() % 100; // Генерация случайных чисел от 0 до 99

}


//printf("Original array:\n");

//for (int i = 0; i < arraySize; i++) {

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

//}

//printf("\n");


struct timeval start, end;

gettimeofday(&start, NULL);


timSort(arr, arraySize, maxThreads);


gettimeofday(&end, NULL);


//printf("Sorted array:\n");
```

```
//for (int i = 0; i < arraySize; i++) {

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

//}

//printf("\n");



long seconds = end.tv_sec - start.tv_sec;

long microseconds = end.tv_usec - start.tv_usec;

double elapsed = seconds + microseconds * 1e-6;



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

printf("Number of threads used: %d\n", maxThreads);



free(arr);

return 0;

}
```

Протокол работы программы

Тестирование:

apple@MacBook-Pro-apple src % ./timsort2 4
Enter the number of elements in the array: 10
Original array:
8 48 25 62 61 11 30 30 50 60
Sorted array:
8 11 25 30 30 48 50 60 61 62
Time taken for sorting: 0.000246 seconds
Number of threads used: 4

Strace:

execve("./timsort2", ["./timsort2", "3"], 0xfffff3d44008 /* 9 vars */) = 0
brk(NULL) = 0xaaaafda59000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffb29d3000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=8467, ...}) = 0
mmap(NULL, 8467, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffffb29d0000
close(3)                                = 0
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/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\267\0\1\0\0\0\360\206\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1722920, ...}) = 0
mmap(NULL, 1892240, PROT_NONE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_DENYWRITE, -1, 0) = 0xffffb27cc000
mmap(0xffffb27d0000, 1826704, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0xffffb27d0000
munmap(0xffffb27cc000, 16384)           = 0
munmap(0xffffb298e000, 49040)           = 0
mprotect(0xffffb296a000, 77824, PROT_NONE) = 0
mmap(0xffffb297d000, 20480, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19d000) = 0xffffb297d000
mmap(0xffffb2982000, 49040, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffffb2982000
close(3)                                = 0
set_tid_address(0xffffb29d3fb0)         = 1495
set_robust_list(0xffffb29d3fc0, 24)     = 0
rseq(0xffffb29d4600, 0x20, 0, 0xd428bc00) = 0
mprotect(0xffffb297d000, 12288, PROT_READ) = 0
mprotect(0xaaaacbb4f000, 4096, PROT_READ) = 0
mprotect(0xffffb29d8000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0xffffb29d0000, 8467)            = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
getrandom("\x05\xe8\xac\x89\x05\x92\xc1\x07", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0xaaaafda59000
brk(0xaaaafda7a000)                     = 0xaaaafda7a000
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "Enter the number of elements in "..., 43Enter the number of elements in the array: ) = 43
read(0, 100000
"100000\n", 1024)               = 7
mmap(NULL, 401408, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xffffb276e000
```

rt_sigaction(SIGRT_1, {sa_handler=0xffffb2852840, sa_mask=[], sa_flags=SA_ONSTACK|SA_RESTART|SA_SIGINFO}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
mmap(NULL, 8454144, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xffffb1e00000
mprotect(0xffffb1e10000, 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=0xffffb260f270, parent_tid=0xffffb260f270, exit_signal=0, stack=0xffffb1e00000, stack_size=0x80ea60, tls=0xffffb260f8e0} => {parent_tid=[1496]}, 88) = 1496
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8454144, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xffffb1400000
mprotect(0xffffb1410000, 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=0xffffb1c0f270, parent_tid=0xffffb1c0f270, exit_signal=0, stack=0xffffb1400000, stack_size=0x80ea60, tls=0xffffb1c0f8e0} => {parent_tid=[1497]}, 88) = 1497
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
mmap(NULL, 8454144, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xffffb0a00000
mprotect(0xffffb0a10000, 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=0xffffb120f270, parent_tid=0xffffb120f270, exit_signal=0, stack=0xffffb0a00000, stack_size=0x80ea60, tls=0xffffb120f8e0} => {parent_tid=[1498]}, 88) = 1498
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
futex(0xffffb260f270, FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME, 1496, NULL, FUTEX_BITSET_MATCH_ANY) = 0
futex(0xffffb1c0f270, FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME, 1497, NULL, FUTEX_BITSET_MATCH_ANY) = 0
futex(0xffffb120f270, FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME, 1498, NULL, FUTEX_BITSET_MATCH_ANY) = 0
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffb270f000

```
mmap(NULL, 135168, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xffffb26ee000
munmap(0xffffb270f000, 135168)          = 0
munmap(0xffffb26ee000, 135168)          = 0
mmap(NULL, 270336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xffffb26ee000
brk(0xaaaafda9b000)                     = 0xaaaafda9b000
munmap(0xffffb26ee000, 270336)          = 0
write(1, "Time taken for sorting: 0.019089"..., 41Time taken for sorting: 0.019089 seconds
) = 41
write(1, "Number of threads used: 3\n", 26Number of threads used: 3
) = 26
munmap(0xffffb276e000, 401408)          = 0
lseek(0, -1, SEEK_CUR)                  = -1 ESPIPE (Illegal seek)
exit_group(0)                           = ?
+++ exited with 0 +++
```

Наблюдения

Массив из 10.000.000 элементов.

Число потоков	Время исполнения (с)	Ускорение	Эффективность
1	1.044298	1	1
2	0.557395	1.873	0.936
3	0.403979	2.585	0.861
4	0.329472	3.169	0.792
5	0.299037	3.492	0.698
6	0.268388	3.891	0.648

Ускорение показывает во сколько раз применение параллельного алгоритма уменьшает время решения задачи по сравнению с последовательным алгоритмом. Ускорение определяется величиной $S_N=T_1/T_N$, где T_1 - время выполнения на одном потоке, T_N - время выполнения на N потоках.

Эффективность - величина $E_N = S_N/N$, где S_N - ускорение, N - количество используемых потоков.

Вывод

На основе тестирования программы с разным количеством потоков и объемом данных можно сделать следующие выводы:

1. Многопоточность значительно ускоряет выполнение программы при грамотном распределении нагрузки между потоками.
2. Для максимального ускорения следует выбирать оптимальное количество потоков, которое соответствует вычислительным возможностям компьютера (например, числу ядер процессора) и объему задачи.
3. Избыточное количество потоков может снижать эффективность работы из-за накладных расходов на управление потоками и синхронизацию.

Таким образом, многопоточность является эффективным инструментом для повышения производительности, если её правильно применять в зависимости от аппаратных характеристик и сложности задачи.