Московский авиационный институт
(Национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 3 по курсу «Операционные системы»

Студент: Сысоев Максим Алексеевич

Группа: 8О-207Б

Преподаватель: Е. С. Миронов

Вариант: 5

Дата:

Оценка:

Москва, 2021

Лабораторная работа №3

1. Описание

Данная лабораторная работа будет выполняться в ОС Unix.

Множественные нити исполнения в одном процессе называют *потоками* и это базовая единица загрузки ЦПУ, состоящая из идентификатора потока, счетчика, регистров и стека. Потоки внутри одного процесса делят секции кода, данных, а также различные ресурсы: описатели открытых файлов, учетные данные процесса сигналы, значения `umask`, `nice`, таймеры и прочее.

Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков 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 входящих данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант 5: Отсортировать массив целых чисел при помощи четно-нечетной сортировки Бетчера.

С помощью введенных операций алгоритм формулируется довольно просто. С помощью операции unshuffle мы разбиваем массив на две половины. Далее надо уже отсортировать каждую из этих половин и потом слить обратно с помощью операции shuffle. Алгоритм не просто так называется четно-нечетной сортировкой *слиянием* — подход аналогичен известной *merge sort*, разве что логика разбиения на части другая — по четности индекса, а не просто пополам.

queue.h – очередь для потоков.
queue.c – реализация очереди для потоков.
lab3.c – основной код лабораторной работы.

2. Исходный код

queue.h

```c
#ifndef QUEUE_H_
#define QUEUE_H_

#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct queue queue;

typedef struct list list;

struct queue
{
    list *first;
    list *last;
    int size;
};

struct list
{
    list *next;
    list *prev;
    pthread_t *thread;
};

void create(queue **q);

void push(queue *q, pthread_t *thread);
void pop(queue *q);

bool isEmpty(const queue *q);
size_t sizes(const queue *q);

void deletes(queue *q);


#endif
```

queue.c

```c
#include "queue.h"

void create(queue **q)
{
    *q = malloc(sizeof(queue));
    (*q)->size = 0;
    (*q)->first = (*q)->last = NULL;
}

bool isEmpty(const queue *q)
{
    return q->size == 0;
}
```

```
void pop(queue *q)
{
    if (isEmpty(q)) {
        return;
    }

    if (q->size == 1) {
        free(q->first);
        q->first = q->last = NULL;
    }
    else {
        q->first = q->first->next;
        q->first->prev->next = NULL;
        free(q->first->prev);
        q->first->prev = NULL;
    }
    q->size--;
}

void push(queue *q, pthread_t *pt)
{

    if (isEmpty(q)) {
        q->first = malloc(sizeof(list));
        q->first->next = NULL;
        q->first->prev = NULL;
        q->last = q->first;
        q->first->thread = pt;
    }
    else {
        q->last->next = malloc(sizeof(list));
        q->last->next->prev = q->last;
        q->last->next->next = NULL;
        q->last->next->thread = pt;
        q->last = q->last->next;
    }
    q->size++;
}

size_t sizes(const queue *q)
{
    return q->size;
}

void deletes(queue *q)
{
    if (q == NULL) {
        return;
    }
    while (!isEmpty(q)) {
        pop(q);
    }
}
```

lab3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include <math.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include "queue.h"

struct packet
{
    int *a;
    int size;
    int l;
    int r;
};


void compexch(int *a, int *b)
{
    if (*b < *a) {
        int t = *a;
        *a = *b;
        *b = t;
    }
}

bool is_int(char *a)
{
    char *temp = a;
    if (*temp == '-') temp++;
    while (*temp != '\0') {
        if (*temp > '9' || *temp < '0') {
            return false;
        }
        temp++;
    }
    return true;
}

int str_to_int(char *a)
{
    int sign = 1;
    if (*a == '-') {
        sign = -1;
        a++;
    }
    int res = 0;
    for (char *temp = a; *temp != '\0'; temp++) {
        res *= 10;
        res += *temp - '0';
    }
    if (sign == -1) {
        res *= -1;
    }
    return res;
}
```

```
int max_from_array(int *a, const int size)
{
    int max = INT_MIN;
    for (int i = 0; i < size; i++) {
        if (a[i] > max) max = a[i];
    }
    return max;
}

void *shuffle(void *arg)
{
    struct packet *received = (struct packet *) arg;

    int half = (int) (received->l + received->r) / 2;
    int tmp[received->size];
    int i;
    int win_right = 0;
    int win_left = 0;

    for (i = 0; i < received->size; i++)
        tmp[i] = received->a[i];

    while (win_right + win_left != (received->r - received->l) + 1) {
        if (win_left == (received->r - received->l + 1) / 2) {
            tmp[received->l + win_right + win_left] = received->a[half + 1
+ win_right];
            win_right++;
            continue;
        }
        if (win_right == (received->r - received->l + 1) / 2) {
            tmp[received->l + win_right + win_left] = received->a[received-
>l + win_left];
            win_left++;
            continue;
        }
        if (received->a[received->l + win_left] > received->a[half + 1 +
win_right]) {
            tmp[received->l + win_right + win_left] = received->a[half + 1
+ win_right];
            win_right++;
        }
        else {
            tmp[received->l + win_right + win_left] = received->a[received-
>l + win_left];
            win_left++;
        }
    }
    for (i = 0; i < received->size; i++)
        received->a[i] = tmp[i];

    free(received);
    return NULL;
}

void unshuffle(int a[], const int size, int l, int r)
{
    int half = (int) (l + r) / 2;
```

```
    int tmp[size];
    int i, j;
    for (i = 0; i < size; i++)
        tmp[i] = a[i];
    for (i = l, j = 0; i <= r; i += 2, j++) {
        tmp[l + j] = a[i];
        tmp[half + j + 1] = a[i + 1];
    }
    for (i = 0; i < size; i++)
        a[i] = tmp[i];
}

void *OddEvenSplitThreadEdition(void *arg)
{
    struct packet *received = (struct packet *) arg;
    if (received->r == received->l + 1) compexch(&received->a[received->l],
&received->a[received->r]);
    if (received->r < received->l + 2) {
        free(received);
        return NULL;
    }
    unshuffle(received->a, received->size, received->l, received->r);
    free(received);
    return NULL;
}

void OddEvenSplit(int a[], const int size, const int max_threads, queue *q)
{
    int l = 0;
    int r = size - 1;
    if (r == l + 1) compexch(&a[l], &a[r]);
    if (r < l + 2) return;
    unshuffle(a, size, l, r);
    if(max_threads < 0) {
        for (int i = 0; i < log2(size); i++) {
            int j = pow(2, i);
            l = 0;
            r = size - 1;
            for (int z = 0; z < i; z++) // вычисление начального значения
для r
                r = (int) (l + r) / 2;
            for (int k = 0; k < j; k++) {
                struct packet *s = malloc(sizeof(struct packet));
                s->a = a;
                s->l = l;
                s->r = r;
                s->size = size;
                pthread_t *thr = malloc(sizeof(pthread_t));
                point3:
                {
                    int check = pthread_create(thr, NULL, OddEvenSplit-
ThreadEdition, (void *) s);
                    if (check != 0) {
                        pthread_join(*(q->first->thread), NULL);
                        pop(q);
                        goto point3;
                    }
                    else {
```

```
                        push(q, thr);
                    }
                }
                l += pow(2, log2(size) - i);
                r += pow(2, log2(size) - i);
            }
            while (!isEmpty(q)) {
                pthread_join(*(q->first->thread), NULL);
                pop(q);
            }

        }
    } else{
        for (int i = 0; i < log2(size); i++) {
            int j = pow(2, i);
            l = 0;
            r = size - 1;
            for (int z = 0; z < i; z++) // вычисление начального значения
для r
                r = (int) (l + r) / 2;
            for (int k = 0; k < j; k++) {
                struct packet *s = malloc(sizeof(struct packet));
                s->a = a;
                s->l = l;
                s->r = r;
                s->size = size;
                pthread_t *thr = malloc(sizeof(pthread_t));
                point4:
                {
                    if (sizes(q) < max_threads) {
                        int check = pthread_create(thr, NULL, OddEvenSplit-
ThreadEdition, (void *) s);
                        if (check != 0) {
                            pthread_join(*(q->first->thread), NULL);
                            pop(q);
                            goto point4;
                        }
                        else {
                            push(q, thr);
                        }
                    } else{
                        pthread_join(*(q->first->thread), NULL);
                        pop(q);
                        goto point4;
                    }
                }
                l += pow(2, log2(size) - i);
                r += pow(2, log2(size) - i);
            }
            while (!isEmpty(q)) {
                pthread_join(*(q->first->thread), NULL);
                pop(q);
            }

        }

    }
}
```

```
void sort(int a[], const int size, const int max_threads, queue *q)
{
    OddEvenSplit(a, size, max_threads, q);
    int l, r;
    int i, j;
    if (max_threads < 0) { // no limit
        for (i = 1; i <= log2(size); i++) {
            j = pow(2, i + 1);
            for (l = 0, r = j - 1; r < size; l += j, r += j) {
                struct packet *s = malloc(sizeof(struct packet));
                s->a = a;
                s->l = l;
                s->r = r;
                s->size = size;
                pthread_t *thr = malloc(sizeof(pthread_t));
                point1:
                {
                    int check = pthread_create(thr, NULL, shuffle, (void *)
s);
                    if (check != 0) {
                        pthread_join(*(q->first->thread), NULL);
                        pop(q);
                        goto point1;
                    }
                    else {
                        push(q, thr);
                    }
                }
            }
            while (!isEmpty(q)) {
                pthread_join(*(q->first->thread), NULL);
                pop(q);
            }
        }
    }
    else { // limit
        for (i = 1; i < log2(size); i++) {
            j = pow(2, i + 1);
            for (l = 0, r = j - 1; r < size; l += j, r += j) {
                struct packet *s = malloc(sizeof(struct packet));
                s->a = a;
                s->l = l;
                s->r = r;
                s->size = size;
                pthread_t *thr = malloc(sizeof(pthread_t));
                point2:
                {
                    if (sizes(q) < max_threads) {
                        int check = pthread_create(thr, NULL, shuffle,
(void *) s);
                        if (check != 0) {
                            pthread_join(*(q->first->thread), NULL);
                            pop(q);
                            goto point2;
                        }
                        else {
                            push(q, thr);
```

```
                        }
                    }
                    else {
                        pthread_join(*(q->first->thread), NULL);
                        pop(q);
                        goto point2;
                    }
                }
            }
            while (!isEmpty(q)) {
                pthread_join(*(q->first->thread), NULL);
                pop(q);
            }
        }
    }

}

int fpeek(void)
{
    int c;
    c = fgetc(stdin);
    if (c != EOF && c != '\n') ungetc(c, stdin);
    return c;
}

int *get_array(int *size)
{
    *size = 0;
    int number;
    int capacity = 1;
    int c;
    int *a = (int *) malloc(sizeof(int));
    c = getchar();
    printf("Enter array : \n");
    while ((c = fpeek()) != '\n' && c != EOF) {
        scanf("%d", &number);
        a[(*size)++] = number;
        if (capacity <= *size) {
            capacity *= 2;
            a = (int *) realloc(a, capacity * sizeof(int));
        }
    }
    return a;
}

int *generate_array(int size)
{
    unsigned int range;
    printf("Enter a range of numbers in the array. Max value is %d \n",
INT_MAX);
    scanf("%d", &range);
    int *a = (int *) malloc(size * sizeof(int));
    printf("Your array: ");
    for (int i = 0; i < size; i++) {
        a[i] = rand() % (2 * range + 1) - range;
        printf("%d ", a[i]);
    }
```

```
    printf("\n");
    return a;
}

int main(int argc, char *argv[])
{
    srand(time(NULL));
    int max_threads;

    if (argc == 1) {
        printf("Limit of threads is not set. Default value is 10.\n");
        max_threads = 10;
    }
    else if (argc == 3) {
        if (strcmp(argv[1], "-t") != 0) {
            printf("Error: incorrect key\n");
            printf("Usage: %s [-t MAX_THREADS]\n", argv[0]);
            printf("if MAX_THREADS is negative value, then the number of
threads is unlimited\n");
            return 1;
        }
        if (!is_int(argv[2])) {
            printf("Error: incorrect third argument\n");
            printf("Usage: %s [-t MAX_THREADS]\n", argv[0]);
            printf("if MAX_THREADS is negative value, then the number of
threads is unlimited\n");
            return 1;
        }
        max_threads = str_to_int(argv[2]);
    }
    else {
        printf("Usage: %s [-t MAX_THREADS]\n", argv[0]);
        printf("if MAX_THREADS is negative value, then the number of
threads is unlimited\n");
        return 1;
    }

    char choise;
    int flag = 1;
    int size;
    int max;
    int *a;

    printf("Select an action:\n"
           "1. Read the array\n"
           "2. Generate an array automatically\n");
    choise = getchar();
    while (flag) {
        switch (choise) {
            case '1':a = get_array(&size);
                flag = 0;
                break;
            case '2':printf("Enter the size of array: ");
                scanf("%d", &size);
                a = generate_array(size);
                flag = 0;
                break;
            default:printf("Select 1 or 2 action: ");
```

```
                choise = getchar();
                break;
            }
        }


        int sizeOfArray = pow(2, (int) log2(size) + 1);

        if (size == sizeOfArray / 2) sizeOfArray /= 2;
        else a = (int *) realloc(a, sizeOfArray * sizeof(int));

        max = max_from_array(a, size);
        for (int i = size; i < sizeOfArray; i++)
            a[i] = max;

        queue *q;
        create(&q);

        sort(a, sizeOfArray, max_threads, q);

        printf("Sorted array: \n");
        for (int i = 0; i < size; i++)
            printf("%d ", a[i]);
        printf("\n");

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

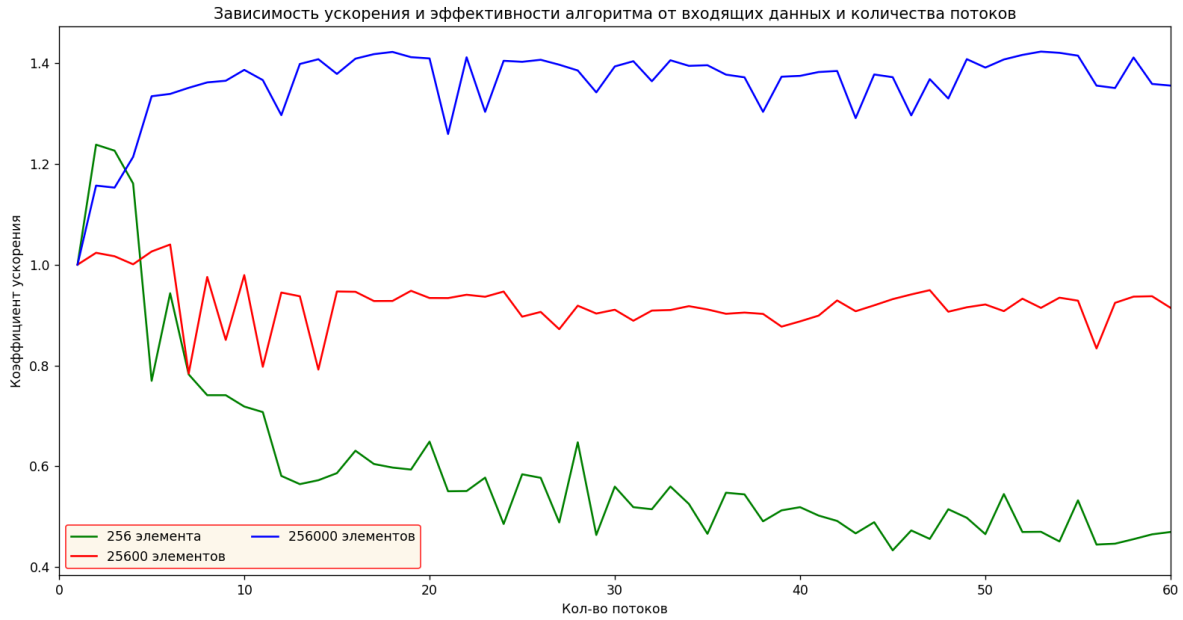
3. Тестирование программы

```
┌──(root💀kali)-[~/MAI_OS/lab3]
└─# ./a.out
Limit of threads is not set. Default value is 10.
Select an action:
1. Read the array
2. Generate an array automatically
1
Enter array :
565 867 545 1
Sorted array:
1 545 565 867
```

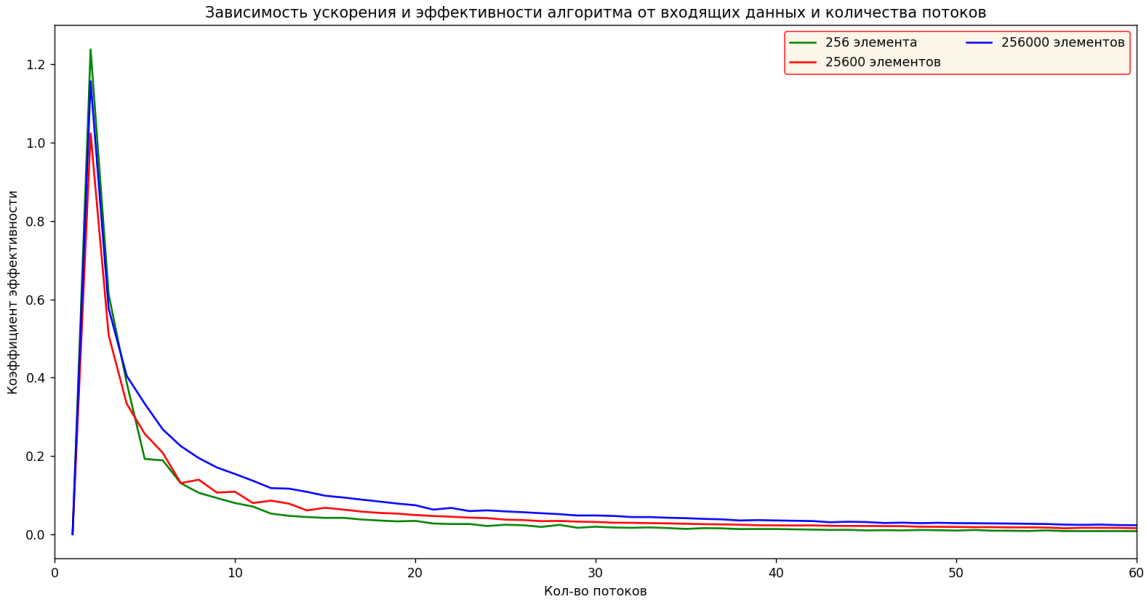
```
┌──(root💀kali)-[~/MAI_OS/lab3]
└─# ./a.out -t 5
Select an action:
1. Read the array
2. Generate an array automatically
2
Enter the size of array:30
Enter a range of numbers in the array. Max value is 2147483647
100
```

```
Your array: 15 98 -16 -71 -41 38 -89 61 46 -84 -28 -41 -77 37 55 84 -9 40 3
50 26 13 -6 -85 41 91 9 76 28 2
Sorted array:
-89 -85 -84 -77 -71 -41 -41 -28 -16 -9 -6 2 3 9 13 15 26 28 37 38 40 41 46
50 55 61 76 84 91 98
```

4. Исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков



Как можно заметить, ускорение повышается при больших данных и падает при маленьких, так как пропорционально много тратиться на системные вызовы.



Как можно заметить, эффективность программы изменяется одинаково при любом количестве входных данных. Она падает. Объяснить это можно тем, что при увеличении кол-ва потоков кол-во системных вызовов увеличивается, а вклад каждого потока падает.

Примечание: *замеры времени производились при помощи утилиты clock().*

5. Вывод

В лабораторной работе удалось поработать с потоками в языке Си. В ходе её выполнения возникало большое количество проблем, которые приходилось решать, переписывая логику программы, например, дедлоки при рекурсивной версии многопоточного деления массива. Работу считаю довольно трудной и от того очень интересной и полезной - множество вещей пришлось предусматривать: от возможных дедлоков, до автоматической генерации входящего массива. Уверен, что опыт в разработке многопоточных программ понадобится в дальнейшем, так как конечному пользователю всегда хочется «быстрее» и при этом, чтобы было правильно – многопоточность, при правильной реализации, способна обеспечить это.