Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ**

Студент: Садаков А. А.

Группа: М8О–206Б–19

Вариант: 4

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

* Создание динамических библиотек
* Создание программ, которые используют функции динамических библиотек

## 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 программ

ой с помощью стандартных средств операционной системы.

В отчёте привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант 4:

Отсортировать массив целых чисел при помощи TimSort

**Общие сведения о программе**

Программа компилируется из файлов main.c, TimSort.h, MTimSort.h. Бенчмарк компилируется из тех же файлов, но вместо main.c используется banchmark.c Также используется заголовочные файлы: stdio.h, unistd.h, stdlib.h, stdbool.h, pthread.h, string.h, sys/time.h. В программе используются следующие системные вызовы:

1. **gettimeofday** ­­–– позволяет получить текущее время. Принимает в качестве аргументов структуру timeval, в которую записывает результат работы, и переменную для коррекции времени. В случае успеха возвращает 0.
2. **pthread\_create ––** (является обёрткой над системным вызовом clone) создаёт новый поток в вызывающем процессе. В качестве аргументов принимает указатель на структуру-идентификатор потока pthread\_t, атрибуты потока, функцию, которая будет запускаться в потоке, список аргументов для функции в виде указателя на void. В случае успеха возвращает 0, иначе возвращает номер ошибки.
3. **pthread\_join ––** используется для ожидания завершения потока. Данная функция блокирует вызывающий поток, пока указанный поток не завершится. В качестве аргументов принимает структуру pthread\_t потока и указатель на переменную, в которую будет записан результат, возвращаемый потоком. В случае успеха возвращает 0, иначе возвращает номер ошибки.
4. **pthread\_exit ––** завершает вызываемый поток. В качестве аргумента принимает значение, которое вернется при завершении потока. Функция всегда завершается успехом.
5. **read ––** предназначена для чтения какого-то числа байт из файла, принимает в качестве аргументов файловый дескриптор, буфер, в который будут записаны данные и число байт. В случае успеха вернет число прочитанных байт, иначе -1.
6. **write ––** предназначена для записи какого-то числа байт в файл, принимает в качестве аргументов файловый дескриптор, буфер, из которого будут считаны данные для записи и число байт. В случае успеха вернёт число записанных байт, иначе -1.

**Общий метод и алгоритм решения**.

Для реализации, поставленной задачи необходимо:

1. Изучить работу с потоками в C.
2. Написать функции слияния массивов, сортировки вставками и реверса.
3. Реализовать функцию, которая будет создаваться в потоках и вызывать функции-помощники.
4. Написать TimSort для однопоточной и многопоточной программы
5. Реализовать бенчмарк для замера времени исполнения однопоточного алгоритма сортировки и многопоточного.

**Основные файлы программы**

**main.c**

#include <stdio.h>

#include "MTimSort.h"

size\_t threads\_count;

int main (int argc, char\*\* argv) {

    if (argc != 2) {

        fprintf(stderr, "wrong number of args\n");

        exit(-1);

    }

    threads\_count = atoi(argv[1]);

    size\_t n;

    scanf("%ld",&n);

    int\* mas = malloc(sizeof(int) \* n);

    int el;

    for (int i = 0; i < n; ++i) {

        if(scanf("%d", &el) != 1) {

            fprintf(stderr, "out of data\n");

            exit(-1);

        }

        mas[i] = el;

    }

    MTimSort(mas, n);

    for (int i = 0; i < n; ++i) {

        printf("%d ", mas[i]);

    }

    printf("\n");

    free(mas);

    return 0;

}

**MTimSort.h:**

#ifndef MTIMSORT\_H

#define MTIMSORT\_H

#include <unistd.h>

#include <pthread.h>

#include "General.h"

extern size\_t threads\_count;

typedef enum RUN\_COMM {

    EXIT = -1,

    INSERT\_SORT,

    REVERSE,

    MERGE,

    PAUSE

} RUN\_COMM;

typedef struct FuncArgs {

    int\* mas;

    size\_t right, mid, left;

    size\_t n;

} FuncArgs;

typedef struct RunArgs {

    RUN\_COMM comm;

    FuncArgs args;

} RunArgs;

void\* Run (void\* data) {

    RunArgs\* a = (RunArgs\*) data;

    while (1) {

        while (a->comm == PAUSE) {}

        switch (a->comm) {

            case INSERT\_SORT:

                InsertSort(a->args.mas, a->args.n);

                break;

            case REVERSE:

                Reverse(a->args.mas, a->args.n);

                break;

            case MERGE:

                Merge(a->args.mas, a->args.left, a->args.mid, a->args.right);

                break;

            case EXIT:

                pthread\_exit(NULL);

                break;

        }

        a->comm = PAUSE;

    }

    pthread\_exit(NULL);

}

void MTimSort (int\* mas, size\_t n) {

    if (n < 64) {

        InsertSort(mas, n);

        return;

    }

    pthread\_t\* tid = malloc(sizeof(pthread\_t) \* threads\_count);

    RunArgs\* Arg = malloc(sizeof(RunArgs) \* threads\_count);

    for (int i = 0; i < threads\_count; ++i) {

        Arg[i].comm = PAUSE;

        if(pthread\_create(&tid[i], NULL, Run, (void\*) &Arg[i]) != 0) {

            fprintf(stderr, "pthread\_create error\n");

            exit(-1);

        }

    }

    size\_t minrun = GetMinRun(n);

    FuncArgs\* arr = malloc(sizeof(FuncArgs) \* (n / 32));

    size\_t arr\_i = 0;

    int index[n], index\_i = 0;

    int i = 0;

    while (i <= n) {

        if (n - i <= 2) {

            if (n - i == 0) {

                break;

            }

            index[index\_i] = n - i;

            ++index\_i;

            InsertSort(mas + i, n - i);

            break;

        }

        int j = i;

        bool(\*Compare)(int, int);

        Compare = mas[j] <= mas[j + 1] ? Compare1 : Compare2;

        while (j + 1 < n && Compare(mas[j], mas[j + 1])) {

            ++j;

        }

        if (Compare == &Compare2) {

            Reverse(mas + i, j - i + 1);

        }

        if (j - i + 1 < minrun) {

            if (n - j - 1 < minrun) {

                j = n - 1;

            } else {

                j = minrun + i - 1;

            }

            arr[arr\_i].mas = mas + i; arr[arr\_i].n = j - i + 1;

            ++arr\_i;

        }

        index[index\_i] = j - i + 1;

        ++index\_i;

        i = j + 1;

    }

    int div = arr\_i / threads\_count, mod = arr\_i % threads\_count;

    for (int j = 0; j < div; ++j) {

        for (int i = 0; i < threads\_count; ++i) {

            Arg[i].args = arr[threads\_count \* j + i];

            Arg[i].comm = INSERT\_SORT;

        }

        for (int i = 0; i < threads\_count; ++i) {

            while (Arg[i].comm != PAUSE) {}

        }

    }

    for (int i = 0; i < mod; ++i) {

        Arg[i].args = arr[div \* threads\_count + i];

        Arg[i].comm = INSERT\_SORT;

    }

    for (int i = 0; i < threads\_count; ++i) {

        while (Arg[i].comm != PAUSE) {}

    }

    size\_t size = index\_i;

    size\_t step = 2;

    while (size > 1) {

        size\_t pairs = size / 2;

        size\_t j = 0;

        arr\_i = 0;

        for (size\_t i = 0; i < pairs; ++i) {

            size\_t i1 = index[i \* step];

            size\_t i2 = index[i \* step + step / 2];

            arr[arr\_i].mas = mas; arr[arr\_i].left = j; arr[arr\_i].mid = j + i1; arr[arr\_i].right = j + i1 + i2;

            ++arr\_i;

            j += i1 + i2;

            index[i \* step] += index[i \* step + step / 2];

        }

        div = pairs / threads\_count, mod = pairs % threads\_count;

        for (int j = 0; j < div; ++j) {

            for (int i = 0; i < threads\_count; ++i) {

                Arg[i].args = arr[threads\_count \* j + i];

                Arg[i].comm = MERGE;

            }

            for (int i = 0; i < threads\_count; ++i) {

                while (Arg[i].comm != PAUSE) {}

            }

        }

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

            Arg[i].args = arr[div \* threads\_count + i];

            Arg[i].comm = MERGE;

        }

        for (int i = 0; i < threads\_count; ++i) {

            while (Arg[i].comm != PAUSE) {}

        }

        arr\_i = 0;

        step \*= 2;

        size = size / 2 + size % 2;

    }

    for (int i = 0; i < threads\_count; ++i) {

        Arg[i].comm = EXIT;

        if (pthread\_join(tid[i], NULL) != 0) {

            fprintf(stderr, "pthread\_join error\n");

            exit(-1);

        }

    }

    free(tid);

    free(Arg);

    free(arr);

}

#endif

**Пример работы**

Тест на корректность сортировки и на время.

**Тест1:**

aleksandr201@LAPTOP-M1VJISD6:~/OS/lab3$ ./a.out 3 < TEST > ans

aleksandr201@LAPTOP-M1VJISD6:~/OS/lab3$ diff ans ANS

aleksandr201@LAPTOP-M1VJISD6:~/OS/lab3$ ./banchmark 3 < TEST

size: 10000

threads: 3

single-threaded: 2ms

multi-threaded: 55ms

**Тест2:**

aleksandr201@LAPTOP-M1VJISD6:~/OS/lab3$ ./a.out 3 < TEST2 > ans

aleksandr201@LAPTOP-M1VJISD6:~/OS/lab3$ diff ans ANS2

aleksandr201@LAPTOP-M1VJISD6:~/OS/lab3$ ./banchmark 3 < TEST2

size: 1000000

threads: 3

single-threaded: 220ms

multi-threaded: 164ms

**Вывод**

Потоки могут как немного ускорить программу, так и замедлить её в несколько десятков раз. Это связано с тем, что потоки создаются не мгновенно, а также с тем, что работу потоков требуется синхронизировать (использовать мьютексы и семафоры). Но, если данные в потоках не пересекаются и если не создавать новые потоки а, например, использовать пул из уже созданных, то можно добиться более быстрого решения задачи.