Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

УПРАВЛЕНИЕ ПОТОКАМИ

Студент: Сектименко Ирина Владимировна
Группа: М8О–210Б–22
Вариант: 18
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023.

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- управлении потоками в ОС;
- обеспечении синхронизации между потоками.

Задание

Составить и отладить программу на языке C/C++, обрабатывающую данные в многопоточном режиме. При обработке использовать стандартные средства создания потоков операционной системы Linux. Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска программы.

Исследовать зависимости ускорения и эффективности алгоритма от входных данных и количества потоков.

Вариант задание – поиск образца в строке наивным алгоритмом.

Общие сведения о программе

Программа компилируется из файла main.cpp (с ключом -pthread). В программе используются следующие системные вызовы:

1. **int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg)** – создание нового потока, причем аргумент *thread является адресом для хранения идентификатора создаваемого потока, start – указатель на потоковую функцию, принимающую бестиповый указатель в качестве единственной переменной, arg – бестиповый указатель, содержащий аргументы потока (чаще всего указывает на глобальную или динамическую переменную), attr – бестиповый указатель атрибутов потока (если он равен NULL, то поток создается с атрибутами по умолчанию).
2. **int pthread_join(pthread_t thread_id, void ** data)** – ожидает завершения потока обозначенного thread_id, если data отличается от NULL, то туда помещаются данные, возвращаемые потоком через функцию pthread_exit() или через инструкцию return потоковой функции.
3. **void pthread_exit(void *retval)** – вызов завершения исполнения потока.

Общий метод и алгоритм решения.

1. Написать наивный алгоритм поиска образца в строке (проходимся по строке, данный символ строки совпадает с первым символом образца, сверяем следующий символ строки и второй символ образца и так до конца образца). Записываем его в потоковую функцию Naive_search.

2. Переменные str, pattern, step, res, которые являются строкой, образцом, длиной части строки, которую будет обрабатывает каждый поток по отдельности, и вектором индексов вхождения образца в строку соответственно, делаем глобальными, чтобы потоки могли получить к ним доступ из потоковой функции.
3. Создаем структуру pthread_args, в которой хранится начало и конец фрагмента, который будет обрабатывать каждый поток.
4. Создаем массив потоков tid и массив их аргументов args_pth. Заполняем последний.
5. С помощью системного вызова pthread_create запускаем каждый поток на своей части строки (mutex не нужен, т к исходные данные мы не изменяем, а только читаем).
6. С помощью системного вызова pthread_join дожидаемся завершения всех потоков.
7. С помощью функции clock() (из “time.h”) высчитываем затраченное время.

Основные файлы программы
main.cpp:

```
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <vector>

std::string str, pattern;
int step;
std::vector<int> res;

struct pthread_args {
    int start;
    int stop;
};

void* Naive_search(void* arg) {
    struct pthread_args* args = ((struct pthread_args*) arg);
    int start = args->start;
```

```
    int stop = args->stop;
    for (int i = start; i < stop; ++i)
    {
        bool flag = true;
        for (int j = 0; j < pattern.size(); ++j) {
            if (i + j >= str.size()) {
                break;
            }
            if (str[i + j] != str[j]) {
                flag = false;
            }
        }
        if (flag) {
            res.push_back(i);
        }
    }
    pthread_exit(0);
}

int main(int argc, char* argv[]) {
    int count_pthreads = 0;
    for (int i = 0; i < static_cast <std::string> (argv[1]).size(); ++i) {
        count_pthreads *= 10;
        count_pthreads += (argv[1][i] - '0');
    }
    std::cin >> str >> pattern;
    step = (str.size()) / count_pthreads;

    clock_t start, end;
    double cpu_time_used;

    pthread_t tid[count_pthreads];
    struct pthread_args *args_pth = (struct pthread_args*) malloc (count_pthreads *
sizeof(struct pthread_args));

    for (int i = 0; i < count_pthreads; ++i) {
        if (i == count_pthreads - 1) {
            args_pth[i].start = i * step;
            args_pth[i].stop = str.size();
        } else {
            args_pth[i].start = i * step;
            args_pth[i].stop = (i + 1) * step;
        }
    }
```

```
    start = clock();
    // struct timespec start, end;
    // clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0; i < count_pthreads; ++i) {
        pthread_create(&tid[i], NULL, Naive_search, (void*)&args_pth[i]);
    }

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

    // clock_gettime(CLOCK_MONOTONIC, &end);
    // long long elapsed_time = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec);
    // printf("Затраченное время для %d потоков: %lld nanoseconds\n", count_pthreads, elapsed_time);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    for (int i = 0; i < res.size(); ++i) {
        std::cout << res[i] << ' ';
    }
    std::cout << std::endl;
    std::cout << cpu_time_used << std::endl;
    return 0;
}
```

Анализ скорости и эффективности

Проведем анализ на нескольких наборах данных.

Возьмем строку из 1000 символов и образец из 3 символов.

Пример входных данных (составлено с помощью генератора, основанного на функции rand из директивы “cstdlib”):

nwlrbbmqbhcdarzowkkyhiddqscdxrjmowfrxsjybldbefsarcbynecdyggxxpklorellnm
papqfwkhopkmcoqhnwnkuewhsqmgbbuqcljjivswmdkqtbxixmvtrrbljptnsnfwzqfjm
afadrrwsofsbcnuvqhffbsaqxwpqcacehchzvfrkmlnozjkpqpxrjxkitzyxacbhhkicqcoen
dtomfgdwdwfcgpxiqvkuytdlcgdewhtaciohordtqkvwcsgspqoqmsboaguwnnyqxnzlg
dgwpbtrwblnsadeuguumoqcdrubetokyxhoachwdvmxxrdryxlmndqtukwagmlejuuk
wcibxubumenmeyatdrmydiajxloghiqfmzhlvihjouvsuyoypayulyeimuotehzriicfskpg
gkbbipzzrzucxamludfykgruowzgiooobppleqlwphapjnadqhdcnvwdtxjbmyppphauxn
spusgdhiixqmbfjxjcvudjsuyibyebmwsiqyoygyxymzevypzvjegebeocfuftsxdixtigsiee
hkchzdflilrjqfnxztqrsvbspkyhsenbppkqtpddbuotbbqcwivrfxjujjddntgeiqvdgaijvwcy

aubwewpjvygehljxepbpiwuqzdzubdubzvafspqpqwuzifwovyddwyvvburczmgyjgfdx
vtnunneslsplwuiupfxlzbknhkwppanltcfirjcddsozoyvegurfwcsfmoxeqmrjowrghwlk
obmeahkgccnaehhsveymqpxhlrnunyfdzrhbasjeuygafoubutpnimuwfjqsjxvkqdorxxv
rwctdsneogvbpkxlpgdirbfcriqifpgynkrrefxsnvucftpwctgtwmxnupycfgcuqunublmoi
itncklefszbexrampetvhqnddjeqvuygpnkazqfrpjvoaxdpcwmjobmskskfojnewxgxnno
fwl
asd

Количество потоков, p	Время поиска с одним потоком, $T1$ (нс)	Время поиска с p потоками, Tp (нс)	Ускорение $Sp = \frac{T1}{Tp}$	Эффективность $Xp = \frac{Sp}{p}$
1	303089	303089	1	1
2	303089	475063	0,64	0,32
3	303089	657976	0,46	0,15
4	303089	396701	0,76	0,19
5	303089	843941	0,36	0,07
6	303089	793373	0,38	0,06
7	303089	1123898	0,27	0,04
8	303089	925490	0,33	0,04
9	303089	1226105	0,25	0,03
10	303089	1538075	0,2	0,02

Проведем анализ на строке из 1000000 символов и образце из трех символов.

Количество потоков, p	Время поиска с одним потоком, $T1$ (нс)	Время поиска с p потоками, Tp (нс)	Ускорение $Sp = \frac{T1}{Tp}$	Эффективность $Xp = \frac{Sp}{p}$
1	24680472	24680472	1	1
2	24680472	13899128	1,78	0,89
3	24680472	10974078	2,25	0,75
4	24680472	7764124	3,18	0,79
5	24680472	8331737	2,96	0,59

6	24680472	8413046	2,93	0,49
7	24680472	9080716	2,72	0,39
8	24680472	9701262	2,54	0,32
9	24680472	9029173	2,73	0,3
10	24680472	10087768	2,45	0,25

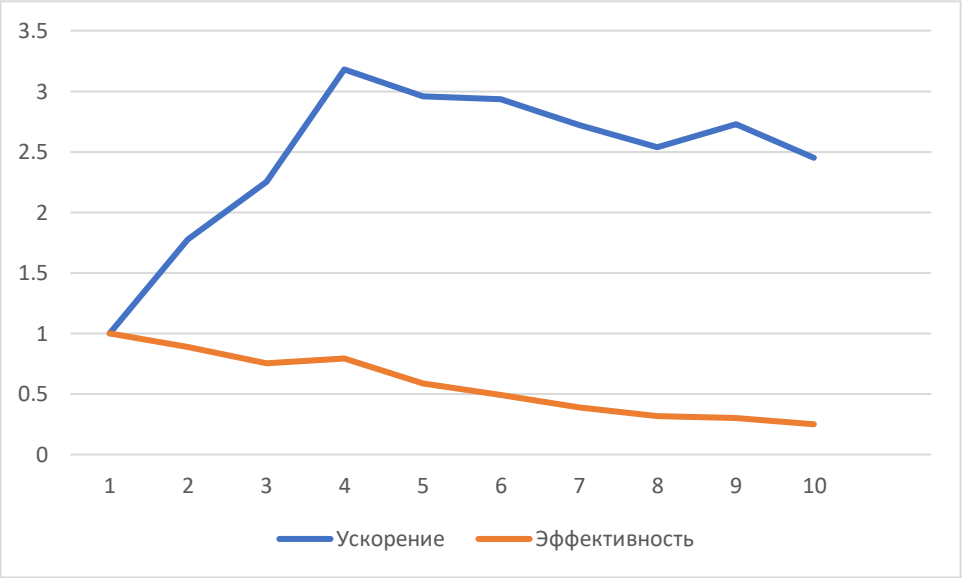

График соответствует второй таблице.

Вывод

Проанализировав полученные в таблицах и на графике данные, можно сказать, что, во-первых, эффективность падает с увеличением потоков, ведь работа с данными и памятью происходит в последовательном режиме, то есть пока один потом получает/изменяет данные, второй ждет. Во-вторых, распараллеливание дает ускорение на больших объемах данных, ведь как раз из-за доступа к данным потокам приходится ждать и на маленьких объемах это очень заметно. В-третьих, на графике заметно, что наибольшее ускорение было достигнуто при 4 потоках. Именно столько ядер находится в процессоре виртуальной машины, и это максимальное число «по-настоящему» параллельных потоков, ведь уже 5-ому, 6-ому и так далее придется «бороться» с остальными за ядра процессора. Поэтому если на компьютере 1 ядро, то бесполезно создавать новые потоки и пытаться распараллелить, это не даст ускорения. В этом я убедилась на собственном опыте, ведь изначально на моей виртуальной машине стояло одно ядро и поэтому при любом количестве потоков время их работы почти не изменялось (а иногда даже ухудшалось).