Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу
«Операционные системы»

Тема работы
“Потоки”

Студент: Прохоров Данила Михайлович
Группа: М8О-208Б-20
Вариант: 2
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/dmprokhorov/OS

Постановка задачи

Задача: реализовать параллельный алгоритм быстрой сортировки.

Общие сведения о программе

В программе используются следующие библиотеки:

- <stdio.h> - для вывода информации на консоль
- <pthread.h> - для работы с потоками
- <stdlib.h> - для вызова функций free и exit
- <limits.h> - для присвоения переменной значения INT_MAX

В задании используются такие команды и строки, как:

- **pthread_create(&threads[number – local_number – 1], NULL, quicksort, &a (или &b)) –** системный вызов, отвечающий за создание потока, принимающий 4 аргумента: указатель на поток, атрибуты по умолчанию, указатель на нужную функцию (откуда поток начнёт своё выполнение) и единственный аргумент этой функции. В случае успеха функция вернёт значение 0, а в случае неуспеха – номер ошибки.
- **pthread_join(&threads[number – local_number – 1], NULL) –** системный вызов, предназначенный для ожидания завершения потока, созданного создающим потоком (в котором прописана данная функция), принимающий 2 аргумента: указатель на созданный поток и указатель на указатель, где будет храниться возвращённое из потока значение. В случае успеха функция вернёт значение 0, а в случае неуспеха – номер ошибки.

- **pthread_mutex_t mutex –** объявление мьютекса mutex в качестве глобальной переменной
- **pthread_mutex_lock(&mutex) –** блокировка мьютекса
- **pthread_mutex_unlock(&mutex) –** разблокировка мьютекса
- **number = strtol(argv[1], NULL, 10) –** функция, преобразовывающая строку в число типа int, принимающая 3 аргумента: строку, ссылку на объект типа char* (значение которой содержит адрес следующего символа в строке, сразу после предыдущего найденного числа, в моём случае указан NULL, то есть этот аргумент игнорируется), и основание системы счисления (в моём случае десятичной).
- **pthread_mutex_init(&mutex, NULL)** – команда, создающая мьютекс и принимающая 2 аргумента: указатель на мьютекс и атрибуты (в моём случае по умолчанию)
- **pthread_mutex_destroy(&mutex)** – команда, уничтожающая мьютекс

Общий метод и алгоритм решения

Я создаю структуру thread_data, хранящую внутри себя индексы i и j (массив с номерами элементов между i и j нужно будет отсортировать), и, собственно, сам массив (целиком).

Далее программа запускается с параметрами n и m (количество потоков и размер массива), если аргументов запуска не 3, количество потоков меньше 0, а размер массива меньше 1 (дробные числа округляются вниз), то программа завершается.

Далее числа n и m сравниваются: если n > m (то есть количество потоков больше размера массива, чего при параллельном алгоритме быстрой сортировки быть не может), то число n становится равным m.

Затем вводятся элементы массива – целые числа, сразу же осуществляется проверка массива на отсортированность, если выясняется, что он уже

отсортирован, то программа выводит сообщение об этом и завершается успешно. В противном случае программа продолжается.
Далее происходит попытка создания мьютекса, если не получилось, то программа завершается.
Потом запускается функция параллельной быстрой сортировки. Она заключается в следующем: после стандартный процедуры обмена элементов относительно заданного (в моём алгоритме это срединный) (то есть при проходе от начала элемента до середины слева направо если элемент больше срединного, то он меняется с элементом, меньшим его, полученном при аналогичном проходе массива от конца до середины справа налево) большая половина массива продолжает сортироваться в текущем потоке, а меньшая, если она есть, отправляется сортироваться в новый поток (если количество созданных потоков не достигло максимума, введённого пользователем). Переменная, отвечающая за количество потоков, которые ещё можно запустить, является глобальной, поэтому когда создаётся новый поток, мьютекс блокирует переменную, она потоком уменьшается, затем мьютекс разблокирует её. Когда переменная стала равна 0, то никаких новых потоков не создаётся, и происходит обычный рекурсивный алгоритм быстрой сортировки в одном потоке.
В функции параллельной быстрой сортировки могут произойти 2 ошибки:
А) Ошибка создания потока, тогда выводится соответствующая информация об этом, и программа аварийно завершается.
Б) Ошибка соединения потоков, тогда также выводится соответствующая информация об этом, и программа аварийно завершается.
Если ошибок обнаружено не было, то сортировка завершается, выводится отсортированный массив, очищается память от массива и созданной структуры, являвшейся аргументом функции. Затем происходит попытка уничтожения мьютекса, если она оказалась неудачной, то выводится соответствующая информация об этом и программа аварийно завершается. В противном случае программа завершается успешно.

Собирается программа при помощи команды gcc task.cc -pthread -o task, запускается при помощи команды ./task n m, где n – количество потоков, используемых программой, n – размер сортируемого массива.

Исходный код

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <limits.h>
unsigned current_number, number;
pthread_t* threads;
pthread_mutex_t mutex;

typedef struct
{
    unsigned i, j;
    int *array;
} thread_data;

void* quicksort (void* arguments)
{
    thread_data* data = (thread_data*) arguments;
    unsigned i = data->i, j = data->j;
    int x = data->array[(i + j) / 2];
    while (i <= j)
    {
        while (data->array[i] < x)
        {
            i++;
        }
        while (data->array[j] > x)
        {
            j--;
        }
        if (i <= j)
        {
            if (data->array[i] > data->array[j])
            {
                data->array[i] += data->array[j];
                data->array[j] = data->array[i] - data->array[j];
                data->array[i] -= data->array[j];
            }
            if (i == INT_MAX)
            {
                break;
            }
            i++;
            if (!j)
            {
                break;
            }
            j--;
```

```
    }
}
if ((i < data->j) && (data->i < j))
{
    thread_data a = {i, data->j, data->array}, b = {data->i, j, data->array};
    if (current_number)
    {
        int index;
        if (index = (pthread_mutex_lock(&mutex)) != 0)
        {
            printf("There is some problems with locking mutex\n");
            printf("Code of error is %d\n", index);
            exit(1);
        }
        current_number--;
        if (index = (pthread_mutex_unlock(&mutex)) != 0)
        {
            printf("There is some priblems with unlocking mutex\n");
            printf("Code of error is %d\n", index);
            exit(1);
        }
        int local_number = current_number;
        if ((data->j - i) >= (j - data->i))
        {
            if (index = (pthread_create(&threads[number - current_number - 1], NULL, quicksort, &b)) != 0)
            {
                printf("Can't create the thread\n");
                printf("Code of error is %d\n", index);
                exit(1);
            }
            quicksort(&a);
        }
        else
        {
            if (index = (pthread_create(&threads[number - current_number - 1], NULL, quicksort, &a)) != 0)
            {
                printf("Can't create the thread\n");
                printf("Code of error is %d\n", index);
                exit(1);
            }
            quicksort(&b);
        }
        if ((index = pthread_join(threads[number - local_number - 1], NULL)) != 0)
        {
            printf("Can't join the thread\n");
            printf("Code of error is %d\n", index);
            exit(1);
```

```
            }
        }
        else
        {
            quicksort(&a);
            quicksort(&b);
        }
    }
    else
    {
        if (i < data->j)
        {
            thread_data a = {i, data->j, data->array};
            quicksort(&a);
        }
        else if (data->i < j)
        {
            thread_data a = {data->i, j, data->array};
            quicksort(&a);
        }
    }
    return NULL;
}

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

```
if ((argc != 3) || (atoi(argv[1]) < 0) || (atoi(argv[2]) < 1))
{
    printf("Syntax should be like this: ./[executable_file_name] [(non-negative) number_of_threads] [(positive) size_of_array]\n");
    exit(1);
}
number = strtol(argv[1], NULL, 10);
unsigned size = strtol(argv[2], NULL, 10);
if (number > size)
{
    printf("The size of array is less than number of threads, but it can't be with parallel quick sort, so number of threads equals size of array now\n");
    number = size;
}
current_number = number;
printf("Input elements of the array\n");
int *array = (int*) malloc(size * sizeof(int));
bool sorted = true;
for (int i = 0; i < size; i++)
{
    scanf("%i", &array[i]);
    if ((i) && (sorted) && (array[i] < array[i - 1]))
    {
```

```
    bool sorted = true;
    for (int i = 0; i < size; i++)
    {
        scanf("%i", &array[i]);
        if ((i) && (sorted) && (array[i] < array[i - 1]))
        {
            sorted = false;
        }
    }
    if (sorted)
    {
        printf("Array is sorted yet, this is the end of the program\n");
        for (int i = 0; i < size; i++)
        {
            printf("%i ", array[i]);
        }
        printf("\n");
        free(array);
        return 0;
    }
    int index;
    if (index = (pthread_mutex_init(&mutex, NULL)) != 0)
    {
        printf("There is some problems with initializing mutex\n");
        printf("%i\n", index);
        free(array);
        exit(1);
    }
    threads = (pthread_t*)malloc(number * sizeof(pthread_t));
    thread_data a = {0, size - 1, array};
    quicksort(&a);
    printf("Sorted array:\n");
    for (int i = 0; i < size; i++)
    {
        printf("%i ", array[i]);
    }
    printf("\n");
    free(array);
    free(threads);
    if (index = (pthread_mutex_destroy(&mutex)) != 0)
    {
        printf("There is some problems with destroying mutex\n");
        printf("%i\n", index);
        exit(1);
    }
    return 0;
}
```

Демонстрация работы программы

Тест 1.

Неправильные аргументы вызова

```
danila@danila-VirtualBox:~/operation_systems/OS3$ ./task -2 -34.5
Syntax should be like this: ./[executable_file_name] [(non-negative) number_of_threads] [(positive) size_of_array]
danila@danila-VirtualBox:~/operation_systems/OS3$
```

Тест 2.

Количество нитей больше количества элементов в массиве

```
danila@danila-VirtualBox:~/operation_systems/OS3$ ./task 8 6
The size of array is less than number of threads, but it can't be with parallel quick sort, so number of threads equals size of array now
Input elements of the array
90 83 -12 1566 2021 5
Sorted array:
-12 5 83 90 1566 2021
danila@danila-VirtualBox:~/operation_systems/OS3$
```

Тест 3.

Массив уже отсортирован

```
danila@danila-VirtualBox:~/operation_systems/OS3$ ./task 4 12
Input elements of the array
1 2 4 8 16 32 64 128 256 512 1024 2048
Array is sorted yet, this is the end of the program
1 2 4 8 16 32 64 128 256 512 1024 2048
danila@danila-VirtualBox:~/operation_systems/OS3$
```

Тест 4.

Программа параллельно сортирует массив из 7 элементов с помощью 4 потоков.

```
danila@danila-VirtualBox:~/operation_systems/OS3$ ./task 4 7
Input elements of the array
10 -52 2746 -67 17 3 43
Sorted array:
-67 -52 3 10 17 43 2746
danila@danila-VirtualBox:~/operation_systems/OS3$
```

Выводы

Благодаря данной лабораторной работе я ознакомился с тем, что из себя представляют потоки в операционной системе Ubuntu. Я узнал некоторые полезные системные вызовы, научился основам пользования мьютексом, а не очень сложное задание лишь помогло мне в этом.