Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу
«Операционные системы»

ПОТОКИ

Студент: Лукманова Аэлита
Группа: М8О–201Б–19
Вариант: 8
Преподаватель: Миронов Е. С.
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021.

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управление потоками в ОС
- Обеспечение синхронизации между потоками

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 8:

Есть K массивов одинаковой длины. Нужно их сложить, предусмотреть стратегию, адаптирующуюся под количество массивов и их длину (по количеству операций).

Общие сведения о программе

Программа компилируется из файла main.c. В программе используются заголовочные файлы: stdio.h, unistd.h, stdlib.h, pthread.h, time.h, dirent.h, sys/time.h, math.h. В программе используются следующие системные вызовы:

1. **clock_t** — работа со временем.
2. **pthread_create** — (является оберткой над системным вызовом clone) создает 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 вернет число записанных байт, иначе -1.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы потоков в C.
2. Реализовать вспомогательные функции для логирования.
3. Реализовать функцию, которая будет запускаться не в главном потоке.
4. Реализовать обработку системных ошибок согласно заданию.

Основные файлы программы

main.h:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <math.h>

struct arg_struct {
    int * x;
    int * total;
    int size;
    int j;
```

```
    int k;
    int * smallArray;
    int numberOfThreads;
};


void* changeTotalArrayByConcat(void *arguments) {

    struct arg_struct *args = (struct arg_struct *)arguments;

    // сколько массивов в переданном smallArray
    int sizeOfSmallArrayColumns;

    //сколько массивов прошло
    int initColumnIndex;
    int r = args->k % args->numberOfThreads;

    if (args->j < r) {
        sizeOfSmallArrayColumns = args->k / args->numberOfThreads + 1;
        initColumnIndex = args->j*sizeOfSmallArrayColumns;
    } else {
        sizeOfSmallArrayColumns = args->k / args->numberOfThreads;
        initColumnIndex = r*(sizeOfSmallArrayColumns+1) + (args->j-r)*sizeOfSmallAr-
rayColumns;
    }

    //uncoment to see work of threads:
    //printf("this is sizeOfSmallArrayColumns: %d\n", sizeOfSmallArrayColumns);

    int *smarr = args->smallArray;

    int indexForTotal = args->size * initColumnIndex;

    int sizeOfSmallArr = args->size * sizeOfSmallArrayColumns;

    memcpy((args->total) + indexForTotal,
           smarr,
           sizeOfSmallArr * sizeof(int));

    //uncoment to see work of threads:
//    for (int i = 0; i < args->size * args->k; i++) {
//
//        printf("f: %d : %d\n", i, (args->total)[i]);
//    }

    free(arguments);
    return NULL;
}

int main(int argc, const char * argv[]) {
//    clock_t start_time;
    int n;
    int k;
    int numberOfThreads;

    printf("How many arrays?:");
    scanf("%d",&k);

    printf("How many elements in array?:");
    scanf("%d",&n);

    printf("How many threads?:");
    scanf("%d",&numberOfThreads);

    //create array of arrays:
    int arrayOfArrays[k][n];

    srand((unsigned int)time( NULL ));

    for (int i = 0; i < k; i++) {
        for (int j = 0; j < n; j++) {
            arrayOfArrays[i][j] = rand()%100 + 1;
```

```
        }
    }

    pthread_t th[k];

    //fill structure:
    struct arg_struct args;
    args.size = n;
    args.k = k;
    args.total = malloc(args.size * args.k * sizeof(int));
    args.numberOfThreads = numberOfThreads;

    clock_t start_time = clock();
    double setUpStructureTime = 0;

    for (int j = 0; j < args.numberOfThreads; j++) {
        clock_t setUpStructureTime0 = clock();
        int x;
        int y = args.size;

        //сколько массивов прошло
        int initColumnIndex;
        //remainder
        int r = args.k % args.numberOfThreads ;

        if (j < r) {
            x = args.k / args.numberOfThreads + 1;
            initColumnIndex = j*x;
        } else {
            x = args.k / args.numberOfThreads;
            initColumnIndex = r*(x+1) + (j-r)*x;
        }

        int smallArray[x][y];

        for (int col = initColumnIndex;
             col < initColumnIndex + x;
             col++) {
            for (int row = 0; row < y; row++) {
                smallArray[col-initColumnIndex][row] = arrayOfArrays[col][row];
            }
        }

        int * smAr = malloc(sizeof(smallArray));
        memcpy(smAr, smallArray, sizeof(smallArray));
        args.smallArray = smAr;
        args.j = j;

        struct arg_struct* tmpArgs = malloc(sizeof(struct arg_struct));
        *tmpArgs = args;


        //printf("setUpStructureTime in %f seconds\n", (double)(clock() - setUpStruc-
tureTime0) / CLOCKS_PER_SEC);
        setUpStructureTime += (double)(clock() - setUpStructureTime0) / CLOCK-
S_PER_SEC;
        pthread_create(&th[j], NULL, &changeTotalArrayByConcat, tmpArgs);

        //uncoment to see work of threads:
        //sleep(1);

    }

    for (int j = 0; j < args.numberOfThreads; j++) {
        if (pthread_join(th[j], NULL) != 0 ) {
            perror("Failed to join thread");
        }
    }
    double elapsed_time = (double)(clock() - start_time) / CLOCKS_PER_SEC - setUp-
StructureTime;
    double elapsed_time2 = (double)(clock() - start_time) / CLOCKS_PER_SEC;
    printf("All Done in %f seconds\n", elapsed_time2);
```

```
    printf("SetUpStructure Done in %f seconds\n", setUpStructureTime);
    printf("Concat Done in %f seconds\n", elapsed_time);

    return 0;

}
```

Пример работы

Тестирование на больших и маленьких случайных тестах:

```
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:1
How many threads?:12
Concat Done in 1.113466
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:1
How many threads?:2
Concat Done in 0.456194
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:2
How many threads?:1
Concat Done in 0.648890
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:2
How many threads?:10
Concat Done in 0.976780
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:2
How many threads?:5
Concat Done in 0.669840
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:1
How many threads?:1
Concat Done in 0.376686
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10
How many elements in array?:1
How many threads?:5
Concat Done in 0.678956
```

```
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10000
How many elements in array?:10
How many threads?:1
Concat Done in 1.208476
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10000
How many elements in array?:10
How many threads?:2
Concat Done in 0.396248
aelitalukmanova@MacBook-Pro-Aelita os-3FF % ./main
How many arrays?:10000
How many elements in array?:10
How many threads?:4
Concat Done in 0.336384
```

Вывод

Потоки очень полезны в случаях, когда большую задачу можно разбить на множество более мелких задач, которые могут выполняться параллельно. Особенно многопоточное программирование начало развиваться с появлением многоядерных процессоров. Однако такое распараллеливание задачи также требует переработки стандартного алгоритма, большей внимательности к совместно-используемым данным, необходимости организовывать общение потоков между собой. Исходя из замеров времени работы алгоритма на маленьких случайных данных, можно сделать вывод, что потоки – излишество на меленьких данных, так как они не дают почти никакого прироста в производительности, а наоборот могут замедлить код, так как создание потоков и их ожидание, переключение контекста тоже занимает какое-то время. На больших случайных данных потоки дают некоторый выигрыш: примерно в два раза между однопоточной программой и двухпоточной программой, и совсем небольшой между двухпоточной программой и четырехпоточной программой. Уменьшение прироста производительности можно объяснить законом Амдала.