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

Лабораторная работа №0 по курсу

«Операционные системы»

Тема работы

“Потоки”

Студент: Зинин Владислав Владимирович
Группа: М8О-208Б-20
Вариант:13
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/frankeloff/OS

Постановка задачи

Задача: Есть набор 128 битных чисел, записанных в шестнадцатеричном представлении, хранящихся в файле. Необходимо посчитать их среднее арифметическое. Округлить результат до целых. Количество используемой оперативной памяти должно задаваться "ключом"

Общие сведения о программе

Для реализации поставленной задачи нам нужны следующие библиотеки:

<unistd.h> - для работы с системными вызовами в Linux

<limits.h> - для определения характеристик общих типов переменных.

<stdlib.h> - для того, чтобы можно было пользоваться функциями, отвечающими за работу с памятью.

<time.h> - для функций, работающих со временем (нужно для строчки srand(time(NULL)) - для генерации рандомных чисел с использованием текущего времени).

<pthread.h> - для работы с потоками.

<ctype.h> - для классификации и преобразования отдельных символов.

<sys/stat.h> - для доступа к файлам.

<fcntl.h> - для работы с файловым дескриптором.

<inttypes.h> - макросы для использования с функциями printf и scanf.

<string.h> - для использования функций над строками.

Для работы с потоками я использую такие системные вызовы, как pthread_create, отвечающий за создание потока, которая в случае успешного выполнения функция возвращает 0. Если произошли ошибки, то могут быть возвращены следующие значения:

- **EAGAIN** – у системы нет ресурсов для создания нового потока, или система не может больше создавать потоков, так как количество

потоков превысило значение PTHREAD_THREADS_MAX (например, на одной из машин, которые используются для тестирования, это магическое число равно 2019)

- **EINVAL** – неправильные атрибуты потока (переданные аргументом attr)
- **EPERM** – Вызывающий поток не имеет должных прав для того, чтобы задать нужные параметры или политики планировщика.

Сигнатура pthread_create следующая: **int pthread_create(*ptherad_t, const pthread_attr_t *attr, void* (*start_routine)(void*), void *arg);**
Где функция получает в качестве аргументов указатель на поток, переменную типа pthread_t, в которую, в случае удачного завершения сохраняет id потока. pthread_attr_t – атрибуты потока. В случае если используются атрибуты по умолчанию, то можно передавать NULL. start_routin – это непосредственно та функция, которая будет выполняться в новом потоке. arg – это аргументы, которые будут переданы функции. Также я использую pthread_join, отвечающий за ожидание завершения потока, имеющий тип возвращаемого значения int и принимающий 2 аргумента: указатель на поток и указатель на указатель в качестве аргумента для хранения возвращаемого значения.

Помимо системных вызовов, связанных с потоками, в моей программе имеются следующие системные вызовы:
off_t lseek(...) - устанавливает смещение для файлового дескриптора в значение аргумента offset.
int open(...) - открытие файлового дескриптора.
void exit(...) - выход из процесса с заданным статусом.
int close(...) - закрытие файлового дескриптора.
Программа собирается и запускается при помощи следующих команд:

make

./generator.exe filename count (например, ./generator.exe nums 100)

./main.exe filename thread_number memory_amount (пример: ./main.exe nums 2 300).

Общий метод и алгоритм решения

Программа на вход получает имя файла, в котором лежат необходимые нам числа (файл так же создается через определенную программу, написанную нами, которая генерирует набор случайных 128-битных чисел), а также количество потоков и количество памяти для них. Сначала мы делаем проверку, хватит ли нам заданной памяти для создания потоков или нет. Если нет, то завершаем программу. Также мы делаем проверку на то, слишком ли много потоков создано для данного количества чисел. Если да, зовершаем программу. Далее мы создаем две структуры, одна хранит в себе массив из потоков, а другая локальные данные для каждого из потоков. Каждый поток подсчитывает свою локальную сумму, после чего в конце работы программы локальные суммы складываются и подсчитывается среднее арифметическое.

Исходный код

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <inttypes.h>
#include <limits.h>
#include <string.h>
#include <ctype.h>

#define INT128LEN 40
#define HEXLEN 32

typedef unsigned __int128 uint128_t;

typedef struct _S_ThreadData {
    const char *filename;

    uint128_t local_sum;
```

```
    size_t nums_count;
    size_t start_pos;
    size_t counts;

} thread_data_t;

typedef struct _S_Args {
    const char *filename;
    size_t threads_num;
    size_t memory_set;
} Args;

typedef struct _S_ThreadArray {
    pthread_t **thread;
    size_t size;
} thread_array_t;

typedef struct _S_ThreadDataArray {
    thread_data_t **thread_data;
    size_t size;
} thread_data_array_t;

int hex_to_dec(char *s)
{
    if(*s == 'A')
        return 10;
    if(*s == 'B')
        return 11;
    if(*s == 'C')
        return 12;
    if(*s == 'D')
        return 13;
    if(*s == 'E')
        return 14;
    if(*s == 'F')
        return 15;
    return 0;
}

uint128_t ato128int(char *str)
{
    uint128_t res = 0;
    while (*str) {
        if(isdigit(*str))
            res = res * 16 + (*str - '0');
        else {
            int convert = hex_to_dec(str);
            res = res * 16 + convert;
        }
        ++str;
    }
    return res;
}

void *thread_function(void *ptr)
{
    thread_data_t *thread_data = (thread_data_t *)ptr;

    char buf[HEXLEN + 1];

    int fd = open(thread_data->filename, O_RDONLY);
    lseek(fd, thread_data->start_pos, SEEK_SET);

    char c;
    for (int i = 0; i < thread_data->nums_count; i++)
    {
        read(fd, buf, HEXLEN);
        buf[HEXLEN] = '\0';

        uint128_t num = ato128int(buf);
        thread_data->local_sum += num;

        read(fd, &c, 1);
        if (c != '\n' && c != '\0') {
            fprintf(stderr, "Num format error\n");
            exit(EXIT_FAILURE);
        }
    }
    close(fd);
    thread_data->local_sum = thread_data->local_sum / thread_data->counts;
```

```
    return 0;
}

void print_int128(uint128_t u128)
{
    char buf[INT128LEN + 1] = {'0'};
    buf[INT128LEN] = '\0';

    int i;
    for (i = INT128LEN - 1; u128 > 0; --i) {
        buf[i] = (int) (u128 % 10) + '0';
        u128 /= 10;
    }

    if (i == INT128LEN) {
        printf("%d\n", 0);
    } else {
        printf("%s\n", &buf[i + 1]);
    }
}

void arg_parse(int argc, const char **argv, Args *args)
{
    if (argc != 4) {
        fprintf(stderr, "Usage: %s [filename] [threads_number] [memory_set]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    args->filename = argv[1];
    args->threads_num = atoi(argv[2]);
    args->memory_set = atoi(argv[3]);
}

void clear_thread_array_t(thread_array_t *cl)
{
    for(int i = 0; i < cl->size; i++)
    {
        free(cl->thread[i]);
    }
    free(cl->thread);
    free(cl);
}

void clear_thread_data_array_t(thread_data_array_t *cl)
{
    for(int i = 0; i < cl->size; i++)
    {
        free(cl->thread_data[i]);
    }
    free(cl->thread_data);
    free(cl);
}

thread_array_t *thread_array_init(size_t thread_num, thread_data_array_t *data_array)
{
   thread_array_t *thread_array = (thread_array_t*) malloc(sizeof(thread_array_t));
   thread_array->thread = (pthread_t**) malloc(sizeof(pthread_t*) * thread_num);
   for (int i = 0; i < thread_num; i++)
   {
        thread_array->thread[i] = (pthread_t*) malloc(sizeof(pthread_t));
        pthread_create(thread_array->thread[i], NULL, thread_function, (void *) data_array->thread_data[i]); //last null is param
   }
   thread_array->size = thread_num;
   return thread_array;
}

thread_data_array_t *thread_array_data_init(size_t thread_num, const char *filename, size_t total_nums, size_t nums_per_thread)
```

```c
{
    thread_data_array_t *th_data_array = (thread_data_array_t*) malloc(sizeof(thread_data_array_t));
    th_data_array->thread_data = (thread_data_t**) malloc(sizeof(thread_data_t*) * thread_num);

    for (int i = 0; i < thread_num; i++)
    {
        thread_data_t *th_data = (thread_data_t*) malloc(sizeof(thread_data_t));
        th_data->local_sum = 0; // количество чисел которое должен считать поток
        th_data->start_pos = i * ((HEXLEN + 1) * nums_per_thread); // Стартовая позиция (на кадой строчке кроме числа есть знак \n)
        th_data->filename = filename;
        th_data->counts = total_nums;
        th_data->nums_count = nums_per_thread;

        th_data_array->thread_data[i] = th_data;
    }
    th_data_array->thread_data[thread_num - 1]->nums_count += total_nums % thread_num; // если вдруг остались числа

    th_data_array->size = thread_num;

    return th_data_array;

}

int main(int argc, const char **argv)
{
    Args args;
    arg_parse(argc, argv, &args); // Парсим аргументы командной строки

    if (args.threads_num * sizeof(thread_data_t) + args.threads_num * sizeof(pthread_t) > args.memory_set) { // Проверям количество опера
        fprintf(stderr, "Too much threads for this amount of memory\n");
        exit(EXIT_FAILURE);
    }
    int fd = open(args.filename, O_RDONLY); // Открываем файл с числами
    size_t file_size = lseek(fd, 0, SEEK_END); // lseek - смещение
    close(fd);

    size_t nums_count = file_size / (HEXLEN + 1);
    size_t nums_per_thread = nums_count / args.threads_num;
    printf("Nums in file: %i\n", nums_count);
    printf("Nums per thread: %i\n", nums_per_thread);
    if (nums_per_thread == 0)
    {
        printf("Too much thread num (%li) for current nums count (%li)\n", args.threads_num, nums_count);
        exit(EXIT_FAILURE);
    }

    thread_data_array_t *thread_data_array = thread_array_data_init(args.threads_num, args.filename, nums_count, nums_per_thread);
    thread_array_t *thread_array = thread_array_init(args.threads_num, thread_data_array);

    for (int j = 0; j < thread_array->size; j++) {
        pthread_join(*thread_array->thread[j], NULL);
    }

    uint128_t sum = 0;
    for (int i = 0; i < thread_data_array->size; i++) {
        sum += thread_data_array->thread_data[i]->local_sum;
    }

    print_int128(sum);

    clear_thread_array_t(thread_array);
    clear_thread_data_array_t(thread_data_array);

    return 0;
}
```

Демонстрация работы программы

Тест 1:

```
→src ./generator.exe nums 100
→src ./main.exe nums 2 20
Too much threads for this amount of memory
```

Тест 2:

```
→src ./generator.exe nums 100
→src ./main.exe nums 2 300
Nums in file: 100
Nums per thread: 50
55581792780935397485137117744421535458
```

Тест 3:

```
→src ./generator.exe nums 150
→src ./main.exe nums 5 600
Nums in file: 150
Nums per thread: 30
48713915612783428231915843976299949 71
```

Выводы

Благодаря данной лабораторной я успешно ознакомился с работой потоков в Linux и тем, как они устроены. Во время выполнения своего задания я изучил многие системные вызовы и научился применять их в своей программе, а также я узнал многие тонкости работы с потоками.