Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу

«Операционные системы»

УПРАВЛЕНИЕ ПОТОКАМИ В ОС

Студент: Лошманов Юрий Андреевич
Группа: М8О–206Б–20
Вариант: 15
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2020.

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.

Перемножение полиномов. На вход подается N-полиномов, необходимо их перемножить

Общие сведения о программе

Программа получает на вход N-полиномов, которые записаны в виде коэффициентов. То есть полученная запись «4 0 2 1 0» эквивалентна «$x^4 + 0x^3 + 2x^2 + 1x + 0$». Программа заполняет динамический массив структурами, представляющими собой многочлен.
Если программа не ограничена пользователем в количестве потоков, то она выбирает их количество, равное количеству потоков, поддерживаемым процессором, умноженное на 8. Если программа ограничена в количестве потоков, то количество потоков ограничивается этим числом, если оно меньше.
Далее создаются потоки и делят поровну между собой динамический массив. Каждый поток перемножает свои элементы, результат перемножения будет самым последним элементом из отведённых потоку, а все остальные будут удалены, для экономии памяти.
Если потоков было несколько, то происходит перемножение оставшихся в массиве многочленов, это тоже происходит в многопоточном режиме, так как многочлены, скорее всего, будут очень больших размеров. Создаются потоки, каждый из которых перемножает два соседних элемента. И так до тех пор, пока не останется один единственный многочлен.

Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Понять различия процессов и потоков
2. Изучить принципы работы функций pthread_create, pthread_join
3. Подключить библиотеку pthread.h, необходимую для работы с вышеперечисленными функциями и их аргументами
4. Реализовать структуру многочлена и функции работы с ней
5. Реализовать динамический массив
6. Написать функции для потоков, одна из которых перемножает два многочлена, а вторая определённый диапазон дин. массива
7. Написать функцию перемножения всех многочленов дин. массива

Основные файлы программы

polynomial.h:

```
#ifndef LAB3_POLYNOMIAL_H
#define LAB3_POLYNOMIAL_H


#include <stdio.h>
#include <stddef.h>
#include <stdint.h>


typedef struct vector vector_t;


typedef struct polynomial {
    size_t degree;
    int64_t *data;
} polynomial_t;


vector_t *parse_polynomials(FILE *input);
polynomial_t *polynomial_create(size_t size);
int polynomial_insert_digit(polynomial_t **polynomial, int64_t digit);
polynomial_t *multiply_polynomials(polynomial_t *a, polynomial_t *b);
int print_polynomial(polynomial_t **p);
void polynomial_destroy(polynomial_t *polynomial);


#endif //LAB3_POLYNOMIAL_H
```

polynomial.c:

```
#include <stdlib.h>
#include <string.h>

#include "polynomial.h"
#include "vector.h"


#define DATA_SIZE 1024


vector_t *parse_polynomials(FILE *input) {
    vector_t *vector = vector_create();

    if (!vector) {
        return NULL;
    }

    char buff[DATA_SIZE];
    while (1) {
        for (int i = 0; i < DATA_SIZE; i++) {
            buff[i] = '\0';
        }
        if (fscanf(input, "%[^\n]\n", buff) == EOF) {
            break;
        }
        char *start = buff;
        char *end;
        polynomial_t *polynomial = polynomial_create((size_t)(DATA_SIZE / 2));
        if (!polynomial) {
            return NULL;
        }
        while (1) {
            int64_t d = strtoull(start, &end, 0);
            if (polynomial_insert_digit(&polynomial, d) != 0) {
                return NULL;
            }
            if (!*end || strcmp(end, " ") == 0) {
                break;
            }
            char *tmp = end;
            end = start;
            start = tmp;

        }
        vector_push_back(&vector, polynomial);
    }
    return vector;
}

polynomial_t *polynomial_create(size_t size) {
    polynomial_t *polynomial = (polynomial_t *)malloc(sizeof(polynomial_t));

    if (!polynomial) {
        return NULL;
    }

    polynomial->degree = 0;
    polynomial->data = (int64_t *)malloc(sizeof(int64_t) * size);

    if (!polynomial->data) {
        return NULL;
    }

    for (size_t i = 0; i < size; i++) {
        polynomial->data[i] = 0;
    }

    return polynomial;
}
```

```
int polynomial_insert_digit(polynomial_t **polynomial, int64_t digit) {
    if ((*polynomial)->degree == DATA_SIZE) {
        return -1;
    }

    (*polynomial)->data[(*polynomial)->degree++] = digit;

    return 0;
}

polynomial_t *multiply_polynomials(polynomial_t *a, polynomial_t *b) {
    int m = a->degree, n = b->degree;
    polynomial_t *result = polynomial_create(m + n - 1);

    if (!result) {
        return NULL;
    }

    result->degree = m + n - 1;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            result->data[i + j] += (a->data[i]) % INT32_MAX * (b->data[j]) %
INT32_MAX;
        }
    }

    return result;
}


int print_polynomial(polynomial_t **polynomial) {
    if (polynomial == NULL || (*polynomial) == NULL || (*polynomial)->data == NULL)
{
        return -1;
    }

    for (size_t i = 0; i < (*polynomial)->degree; i++) {
        int64_t value = (*polynomial)->data[i];
        printf("%llu ", value);

    }
    printf("\n");

    return 0;
}


void polynomial_destroy(polynomial_t *polynomial) {
    free(polynomial->data);
    free(polynomial);
}
```

vector.h:

```
#ifndef LAB3_VECTOR_H
#define LAB3_VECTOR_H


#include "polynomial.h"


typedef polynomial_t* vector_type_t;


typedef struct vector {
    size_t size;
    size_t capacity;
    vector_type_t *data;
} vector_t;
```

```
vector_t *vector_create(void);
int vector_push_back(vector_t **vector, vector_type_t value);
int vector_destroy(vector_t *vector);


#endif //LAB3_VECTOR_H
```

vector.c:

```
#include <stdlib.h>


#include "vector.h"


vector_t *vector_create(void) {
    vector_t *vector = (vector_t *)malloc(sizeof(vector_t));
    if (vector == NULL) {
        fprintf(stderr, "malloc error\n");
        return NULL;
    }

    vector->size = 0;
    vector->capacity = 1000;
    vector->data = (vector_type_t *)malloc(sizeof(vector_type_t) * vector-
>capacity);
    if (!vector->data) {
        fprintf(stderr, "malloc error\n");
        return NULL;
    }

    return vector;
}


int vector_push_back(vector_t **vector, vector_type_t value) {
    if ((*vector)->size == (*vector)->capacity) {
        (*vector)->capacity *= 2;
        (*vector)->data = (vector_type_t *)realloc((*vector)->data,
sizeof(vector_type_t) * (*vector)->capacity);
        if (!(*vector)->data) {
            fprintf(stderr, "realloc error\n");
            return -1;
        }
    }
    (*vector)->data[(*vector)->size++] = value;
    return 0;
}


int vector_destroy(vector_t *vector) {
    free(vector->data);
    free(vector);
    return 0;
}
```

main.c:

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

```
#include "vector.h"
#include "polynomial.h"


long threads_count;


struct args {
    vector_t **vector;
    size_t start;
    size_t finish;
};


void *multiply_part_polynomials(void *args) {
    vector_t **vector = ((struct args *)args)->vector;
    size_t start = ((struct args *)args)->start;
    size_t finish = ((struct args *)args)->finish;

    for (size_t i = start; i + 1 < finish ; i++) {
        polynomial_t *tmp = multiply_polynomials((*vector)->data[i], (*vector)-
>data[i + 1]);
        if (!tmp) {
            fprintf(stderr, "multiply_polynomials error\n");
            return (void *)-1;
        }
        polynomial_destroy((*vector)->data[i]);
        polynomial_destroy((*vector)->data[i + 1]);
        (*vector)->data[i] = NULL;
        (*vector)->data[i + 1] = tmp;
    }

    free(args);
    return NULL;
}


void *multiply_polynomials_args(void *args) {
    vector_t **vector = ((struct args *)args)->vector;
    size_t i = ((struct args *)args)->start;
    size_t j = ((struct args *)args)->finish;

    polynomial_t *tmp = multiply_polynomials((*vector)->data[i], (*vector)-
>data[j]);
    if (!tmp) {
        fprintf(stderr, "multiply_polynomials error\n");
        return (void *)-1;
    }
    polynomial_destroy((*vector)->data[i]);
    polynomial_destroy((*vector)->data[j]);
    (*vector)->data[i] = NULL;
    (*vector)->data[j] = tmp;

    free(args);
    return NULL;
}


polynomial_t *multiply_polynomials_in_vector(vector_t **vector) {
    size_t polynomials_per_thread = (*vector)->size / (size_t)threads_count;
    if (polynomials_per_thread < 2) {
        polynomials_per_thread = 2;
        threads_count = (long)((*vector)->size / (size_t)polynomials_per_thread);
    }

    printf("Using threads = %ld\n", threads_count);

    pthread_t threads[threads_count - 1];

    // launching threads
    for (long i = 0; i < threads_count - 1; i++) {
```

```
        struct args *args = (struct args *)malloc(sizeof(struct args));
        if (!args) {
            fprintf(stderr, "malloc error\n");
            return NULL;
        }
        args->vector = vector;
        args->start = i * polynomials_per_thread;
        args->finish = (i + 1) * polynomials_per_thread;
        if (pthread_create(&threads[i], NULL, multiply_part_polynomials, args) !=
0) {
            fprintf(stderr, "pthread_create error\n");
            return NULL;
        }
    }

    // main thread
    {
        struct args *args = (struct args *) malloc(sizeof(struct args));
        if (!args) {
            fprintf(stderr, "malloc error\n");
            return NULL;
        }

        args->vector = vector;
        args->start = (threads_count - 1) * polynomials_per_thread;
        args->finish = (*vector)->size;
        if (multiply_part_polynomials(args) != NULL) {
            fprintf(stderr, "multiply_part_polynomials error\n");
            return NULL;
        }
    }

    // waiting threads
    for (long i = 0; i < threads_count - 1; i++) {
        void *status;
        pthread_join(threads[i], &status);
        if (status != (void *)0) {
            fprintf(stderr, "multiply_part_polynomials error\n");
            return NULL;
        }
    }

    // multithreading multiplication
    while (threads_count > 1) {
        size_t current_threads_count = 0;
        for (size_t i = 0; i < threads_count; i++) {
            int flag = 0;

            size_t j1 = (i + 1) * polynomials_per_thread - 1;
            size_t j2 = (i + 2) * polynomials_per_thread - 1;

            if (j2 > (*vector)->size - 1) {
                j2 = (*vector)->size - 1;
                flag = 1;
            }

            if (i == threads_count - 1) {
                j1 = j2 = (*vector)->size - 1;
                flag = 1;
            }

            if (i == threads_count - 2) {
                j2 = (*vector)->size - 1;
                flag = 1;
            }

            if (j2 + polynomials_per_thread > (*vector)->size - 1) {
                j2 = (*vector)->size - 1;
                flag = 1;
            }

            if (j1 != j2) {
```

```
                struct args *args = (struct args *)malloc(sizeof(struct args));

                if (!args) {
                    fprintf(stderr, "malloc error\n");
                    return NULL;
                }

                args->vector = vector;
                args->start = j1;
                args->finish = j2;
                if (pthread_create(&threads[current_threads_count], NULL,
multiply_polynomials_args, args) != 0) {
                    fprintf(stderr, "pthread_create error\n");
                    return NULL;
                }
                current_threads_count++;
            }
            if (flag) {
                break;
            }
            i++;
        }

        for (size_t i = 0; i < current_threads_count; i++) {
            void *status;
            pthread_join(threads[i], &status);
            if (status != (void *)0) {
                fprintf(stderr, "multiply_polynomials_args error\n");
                return NULL;
            }
        }

        polynomials_per_thread *= 2;
        if (threads_count % 2 == 1) {
            threads_count++;
        }
        threads_count /= 2;
    }

    // returning result
    polynomial_t *result = (*vector)->data[(*vector)->size - 1];
    (*vector)->data[(*vector)->size - 1] = NULL;
    return result;
}


int main(int argc, char *argv[]) {
    threads_count = sysconf(_SC_NPROCESSORS_ONLN);
    threads_count = (threads_count * threads_count * threads_count) / 2;
    FILE *input = (argc > 2 && strcmp(argv[1], "-f") == 0) ? (fopen(argv[2],
"r")) : (stdin);

    if (!input) {
        fprintf(stderr, "Can't open file");
        exit(1);
    }

    if (argc > 4 && strcmp(argv[3], "-t") == 0) {
        long num = strtol(argv[4], NULL, 0);
        if (num < 1) {
            fprintf(stderr, "Invalid threads count");
            exit(3);
        }
        threads_count = threads_count < num ? threads_count : num;
    }

    vector_t *vector = parse_polynomials(input);

    if (!vector) {
        fprintf(stderr, "parse_polynomials error\n");
        exit(4);
    }
```

```
        polynomial_t *result = multiply_polynomials_in_vector(&vector);

        if (!result) {
            fprintf(stderr, "multiply_polynomials_in_vector error\n");
            exit(5);
        }

        print_polynomial(&result);

        polynomial_destroy(result);
        vector_destroy(vector);

        fclose(input);
        return 0;
    }
```

Пример работы

```
yuryloshmanov@air-urij Desktop % git clone https://github.com/yuryloshmanov/os_lab_3
Клонирование в «os_lab_3»…
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 17 (delta 1), reused 13 (delta 0), pack-reused 0
Распаковка объектов: 100% (17/17), 4.86 KiB | 414.00 KiB/s, готово.
yuryloshmanov@air-urij Desktop % cd os_lab_3/test
yuryloshmanov@air-urij test % ./run.sh
-- The C compiler identification is AppleClang 12.0.0.12000032
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/yuryloshmanov/Desktop/os_lab_3/src
Scanning dependencies of target lab3
[ 25%] Building C object CMakeFiles/lab3.dir/main.c.o
[ 50%] Building C object CMakeFiles/lab3.dir/polynomial.c.o
[ 75%] Building C object CMakeFiles/lab3.dir/vector.c.o
[100%] Linking C executable lab3
[100%] Built target lab3

test1.txt
single-thread time:
real    0m0.005s
user    0m0.001s
sys     0m0.002s

multi-thread time:
real    0m0.007s
user    0m0.001s
sys     0m0.002s

test2.txt
single-thread time:
real    0m0.052s
user    0m0.043s
sys     0m0.004s

multi-thread time:
real    0m0.029s
user    0m0.026s
sys     0m0.006s

test3.txt
single-thread time:
real    0m19.870s
```

```
user    0m19.603s
sys     0m0.114s

multi-thread time:
real    0m6.718s
user    0m11.120s
sys     0m0.094s
yuryloshmanov@air-urij test % ../src/lab3
2 3
4 5
Using threads = 1
8 22 15
yuryloshmanov@air-urij test % ../src/lab3
4 3 5
2
Using threads = 1
8 6 10
yuryloshmanov@air-urij test % ../src/lab3
42 5 2
4 5
2
0
Using threads = 2
0 0 0 0
yuryloshmanov@air-urij test % ./run.sh clean
yuryloshmanov@air-urij test %
```

Вывод

В ходе данной лабораторной работы я понял чем отличаются потоки от процессов, на учился их создавать, ожидать и завершать. Мне пришлось хорошо подумать над алгоритмом перед тем, как её делать. Ведь плохой алгоритм уничтожает преимущество многопоточного программирования. Я очень рад, что мне удалось придумать алгоритм, где у каждого потока чётко разделены обязанности, благодаря тому, они не ждут друг друга и не впадают в критические области.

Выбор такой своеобразной формулы вычисления необходимого числа потоков обусловлен планировщиком моей операционной системы, я замерил на 100000 и более многочленах, и, 32 потока показали максимальную производительность. Многопоточная программа работает до 3х раз быстрее при небольших объёмах многочленов, и до 4х раз быстрее при больших объёмах, чем однопоточная. Это связано с тем, что на моём процессоре 2 ядра, и он поддерживает 4 потока. Потоки можно ограничить аргументом программы, следовательно и ограничить выигрыш по времени.