Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»
Дисциплина «Операционные системы»

Лабораторная работа №3

Тема: Управление потоками в ОС

Студент: Инютин М. А.
Группа: М8О-207Б-19
Преподаватель: Миронов Е. С.
Дата:
Оценка:

Москва, 2020

Постановка задачи

Изучить управление потоками, обеспечение синхронизации между потоками.

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

Алгоритм решения задачи

Полином можно представить как массив пар числе. Первое — коэффициент перед переменной, а второе - степень. Разделим длину N первого полинома на количество потоков K и отправим в каждый поток обрабатывать примерно равные части первого полинома на второй длины M. Представить ответ можно в двух вариантах: создать массив пар размера $N*M$ и избежать синхронизации потоков, создать массив пар размера суммы максимальных степеней, но в таком случае нужно блокировать каждый раз массив, чтобы достичь синхронизации. Я выбрал первый вариант, но для себя сделал второй и сравнил их.

Листинг программы

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

typedef struct TItem {
     int A, B;
} Item;

typedef struct TThreadToken {
     Item* P;
     Item* Q;
     Item* Res;
     int L, R, N, M;
} ThreadToken;

void* ThreadFunc(void* token) {
     ThreadToken* tok = (ThreadToken*) token;
     int l = tok->L;
     int r = tok->R;
     if (l >= tok->N) {
          l = tok->N - 1;
     }
     if (r > tok->N) {
          r = tok->N;
     }
     for (int i = l; i < r; ++i) {
          for (int j = 0; j < tok->M; ++j) {
               tok->Res[tok->M * i + j].A =
                                        tok->P[i].A * tok->Q[j].A;
               tok->Res[tok->M * i + j].B =
                                        tok->P[i].B + tok->Q[j].B;
          }
     }
     return NULL;
}

signed main(int argc, char** argv) {
     int threadNumber = 0;
     for (int i = 0; argv[1][i] > 0; ++i) {
          if (argv[1][i] >= '0' && argv[1][i] <= '9') {
               threadNumber = threadNumber * 10
                                             + argv[1][i] - '0';
          }
     }
     pthread_t* threads = (pthread_t*)malloc(sizeof(pthread_t)
                                                  * threadNumber);
     int n, m;
     scanf("%d%d", &n, &m);
     /* Input p(x) */
     Item* p = malloc(sizeof(Item) * n);
```

```
    for (int i = 0; i < n; ++i) {
        int a, b;
        scanf("%d%d", &a, &b);
        p[i].A = a;
        p[i].B = b;
    }
    /* Input q(x) */
    Item* q = malloc(sizeof(Item) * m);
    for (int i = 0; i < m; ++i) {
        int a, b;
        scanf("%d%d", &a, &b);
        q[i].A = a;
        q[i].B = b;
    }
    Item* res = malloc(sizeof(Item) * (n * m));
    ThreadToken* tokens =
(ThreadToken*)malloc(sizeof(ThreadToken) * threadNumber);
    int step = (n + threadNumber - 1) / threadNumber;
    for (int i = 0; i < threadNumber; ++i) {
        tokens[i].P = p;
        tokens[i].Q = q;
        tokens[i].Res = res;
        tokens[i].L = i * step;
        tokens[i].R = (i + 1) * step + 1;
        tokens[i].N = n;
        tokens[i].M = m;
    }
    for (int i = 0; i < threadNumber; ++i) {
        if (pthread_create(&threads[i], NULL,
                           ThreadFunc, &tokens[i])) {
            printf("Error creating thread!\n");
            return -1;
        }
    }
    for (int i = 0; i < threadNumber; ++i) {
        if (pthread_join(threads[i], NULL)) {
            printf("Error executing thread!\n");
            return -1;
        }
    }
    printf("Execution time %llu ms\n", (end - start));
    for (int i = 0; i < n * m; ++i) {
        printf("%d %d\n", res[i].A, res[i].B);
    }
    free(threads);
    free(p);
    free(q);
    free(res);
    free(tokens);
    return 0;
}
```

Тесты и протокол исполнения

Программа запускается с числовым ключом — числом потоков. Для тестирования корректности программы я использовал 6 потоков. В первой строке записано два числа N и M — размеры полиномов $p(x)$ и $q(x)$ соответственно. В следующих N и M строках записано по паре чисел - коэффициенты перед переменной, и степени переменных для многочленов $p(x)$ и $q(x)$. Программа выводит $N+M$ строк, в каждой из которых пара чисел, так же описывающая член результирующего произведения $p(x)$ и $q(x)$.

Тест №1

Ввод

2 3
1 1
-1 0
1 2
1 1
1 0

Вывод

1 3
1 2
1 1
-1 2
-1 1
-1 0

Тест №2

Ввод

1 5
1 0
5 0
4 1
3 2
2 3
1 4

Вывод

5 0
4 1
3 2
2 3
1 4

Ускорение и эффективность программы

При множественном тестировании я заметил, что код на одном ядре выполняется дольше на последующих тестах. Это связано с устройством ноутбука (недостаток охлаждения). Перед тестированием я каждый раз запускал программу несколько раз, чтобы тесты были более приближены к реальности. Тестирование производилось на полиномах размерами 10^4 .

K	Время исполнения	Ускорение	Эффективность
1	830 ms	1.00	1.00
2	441 ms	1.92	0.96
3	329 ms	2.58	0.86
4	300 ms	2.83	0.71
5	284 ms	2.99	0.60
6	263 ms	3.23	0.54
7	254 ms	3.35	0.58
8	240 ms	3.54	0.44
9	223 ms	3.81	0.42
10	217 ms	3.92	0.42
11	211 ms	4.03	0.37
12	218 ms	3.90	0.32
13	234 ms	3.63	0.28
14	222 ms	3.83	0.27
15	240 ms	3.54	0.23
16	240 ms	3.54	0.22

Видно, что после 11-12 потоков ускорение начало уменьшаться. Мой процессор имеет 6 ядер и 12 потоков. Поэтому удалось достичь максимального ускорения именно при таком количестве потоков K .

Выводы

Я научился работать с потоками на примере POSIX Threads, составил и отладил программу перемножения полиномов. При тестировании мне сильно помог bash скрипт, который сам запускал программу с разным количеством потоков. Во время выполнения работы я понял, что создать и синхронизировать много потоков может быть более накладно, чем выполнять код на одном ядре. Я дополнительно реализовал свой вариант, использующий меньше памяти, но требующий синхронизации. Из-за этого при создании двух и более потоков программа сильно замедлилась (по сравнению с одним потоком), но даже один поток с мьютексом работал на порядок дольше, чем программа без синхронизации. Если задачу в программировании можно выполнять по частям, независимо друг от друга, то многопоточный подход ускорит работу программы (если синхронизации не обойдётся дороже!) в несколько раз. Так в играх вычисления (например, перемножение матриц) производятся на большом количестве графических ядер параллельно.

Список литературы

1. Таненбаум Э., Бос Х. *Современные операционные системы. — 4-е изд.* — СПб.: Издательский дом «Питер», 2018. — С. 123 - 146
2. POSIX Threads — Википедия
   URL: ru.wikipedia.org/wiki/POSIX_Threads (дата обращения 18.10.2020)