Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Лабораторная работа №10

Выполнил: студент группы ИП-211

Оганесян Альберт

Лацук Андрей

Проверил:

Профессор кафедры ПМиК

Малков Е. А.

Новосибирск 2024

**Задание:** протестируйте программы лабораторных 8 и 9, используя

программную реализацию алгоритма Петерсона, запуская их на

одном, двух и нескольких ядрах. Протестируйте модифицированный

на основе атомарных функций код алгоритма Петерсона используя

различные модели упорядочения выполнения инструкций кода.

**Цель:** знакомство с атомарными функциями.

1. Добавим в программу из 8 лабораторной алгоритм Петерсона:

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <time.h>**

**#define ARRAY\_SIZE 1000000**

**#define NUM\_THREADS 2**

**typedef struct**

**{**

**int start;**

**int end;**

**int \*array;**

**long long partial\_sum;**

**} ThreadData;**

**volatile int flag[NUM\_THREADS];**

**volatile int turn;**

**long long total\_sum = 0;**

**void \*calculate\_partial\_sum(void \*arg)**

**{**

**ThreadData \*data = (ThreadData \*)arg;**

**data->partial\_sum = 0;**

**for (int i = data->start; i < data->end; i++)**

**{**

**data->partial\_sum += data->array[i];**

**}**

**int thread\_id = data->start / (ARRAY\_SIZE / NUM\_THREADS);**

**flag[thread\_id] = 1;**

**turn = thread\_id;**

**for (int i = 0; i < NUM\_THREADS; i++)**

**{**

**if (i == thread\_id)**

**continue;**

**while (flag[i] && turn == thread\_id)**

**;**

**}**

**total\_sum += data->partial\_sum;**

**flag[thread\_id] = 0;**

**return NULL;**

**}**

**int main()**

**{**

**int array[ARRAY\_SIZE];**

**pthread\_t threads[NUM\_THREADS];**

**ThreadData thread\_data[NUM\_THREADS];**

**clock\_t start\_time, end\_time;**

**for (int i = 0; i < ARRAY\_SIZE; i++)**

**{**

**array[i] = 1;**

**}**

**for (int i = 0; i < NUM\_THREADS; i++)**

**{**

**flag[i] = 0;**

**}**

**turn = 0;**

**start\_time = clock();**

**int segment\_size = ARRAY\_SIZE / NUM\_THREADS;**

**for (int i = 0; i < NUM\_THREADS; i++)**

**{**

**thread\_data[i].start = i \* segment\_size;**

**thread\_data[i].end = (i == NUM\_THREADS - 1) ? ARRAY\_SIZE : (i + 1) \* segment\_size;**

**thread\_data[i].array = array;**

**pthread\_create(&threads[i], NULL, calculate\_partial\_sum, &thread\_data[i]);**

**}**

**for (int i = 0; i < NUM\_THREADS; i++)**

**{**

**pthread\_join(threads[i], NULL);**

**}**

**end\_time = clock();**

**printf("Сумма элементов массива: %lld\n", total\_sum);**

**printf("Время выполнения: %.6f секунд\n", (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC);**

**return 0;**

**}**

**2.** Запустим программы из 8-й и 9-й лабораторной на разном количестве потоков при помощи команды

taskset --cpu-list \*потоки\* ./lab8

**3.** Заметим, что код лабораторной 8 выдает различные значения при одних и тех же тестах (табл. 3.1). Предположительно, это связано тем, что потоки работают с одним и тем же адресом данных, так как в лабораторной 9, где мы использовали спин-лок такого не наблюдается (табл. 3.2).

| Номер попытки | 8 лабораторная число ядер - 1 | 8 лабораторная число ядер - 2 | 8 лабораторная число ядер - неограниченно |
| --- | --- | --- | --- |
| 1 | 0.001864 | 0.000108 | 0.005218 |
| 2 | 0.001529 | 0.002205 | 0.000205 |
| 3 | 0.000974 | 0.002438 | 0.007018 |
| 4 | 0.000958 | 0.000128 | 0.000161 |
| 5 | 0.001319 | 0.000087 | 0.000185 |
| 6 | 0.000958 | 0.011330 | 0.003736 |
| 7 | 0.001319 | 0.003133 | 0.000161 |
| 8 | 0.001128 | 0.000159 | 0.004488 |
| 9 | 0.001322 | 0.006680 | 0.000138 |
| 10 | 0.001571 | 0.001764 | 0.000185 |

3.1 Таблица со значениями затрат по времени программы по подсчету суммы массива на 1000000 элементах

| Номер попытки | 9 лабораторная число ядер - 1 | 9 лабораторная число ядер - 2 | 9 лабораторная число ядер - неограниченно |
| --- | --- | --- | --- |
| 1 | 102 | 0 | 0 |
| 2 | 101 | 0 | 0 |
| 3 | 97.5 | 0 | 0 |

3.2 Таблица со значениями затрат по времени на 100 итерациях

**4.** Дополним алгоритм Петерсона атомарными функциями:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <signal.h>

#include <string.h>

#include <unistd.h>

#define \_\_RELAXED \_\_ATOMIC\_RELAXED

#define \_\_ACQUIRE \_\_ATOMIC\_ACQUIRE

#define \_\_RELEASE \_\_ATOMIC\_RELEASE

#define \_\_SEQ\_CST \_\_ATOMIC\_SEQ\_CST

volatile int running = 1;

char sh[256];

volatile int flag[2] = {0, 0};

volatile int turn = 0;

void \*Thread(void \*pParams);

void handle\_sigint(int sig)

{

\_\_atomic\_store\_n(&running, 0, \_\_RELAXED);

}

int main(void)

{

pthread\_t thread\_id;

signal(SIGINT, handle\_sigint);

pthread\_create(&thread\_id, NULL, &Thread, NULL);

while (\_\_atomic\_load\_n(&running, \_\_RELAXED))

{

\_\_atomic\_store\_n(&flag[0], 1, \_\_RELAXED);

\_\_atomic\_store\_n(&turn, 1, \_\_RELAXED);

while (\_\_atomic\_load\_n(&flag[1], \_\_RELAXED) && \_\_atomic\_load\_n(&turn, \_\_RELAXED) == 1)

{

}

printf("%s", sh);

fflush(stdout);

\_\_atomic\_store\_n(&flag[0], 0, \_\_RELAXED);

}

pthread\_cancel(thread\_id);

pthread\_join(thread\_id, NULL);

return 0;

}

void \*Thread(void \*pParams)

{

int counter = 0;

while (\_\_atomic\_load\_n(&running, \_\_RELAXED))

{

\_\_atomic\_store\_n(&flag[1], 1, \_\_RELAXED);

\_\_atomic\_store\_n(&turn, 0, \_\_RELAXED);

while (\_\_atomic\_load\_n(&flag[0], \_\_RELAXED) && \_\_atomic\_load\_n(&turn, \_\_RELAXED) == 0)

{

}

if (counter % 2)

{

strcpy(sh, "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");

}

else

{

strcpy(sh, "----------------------------------------------------------------------------------------------------------------------------------\n");

}

counter++;

\_\_atomic\_store\_n(&flag[1], 0, \_\_RELAXED);

}

return NULL;

}

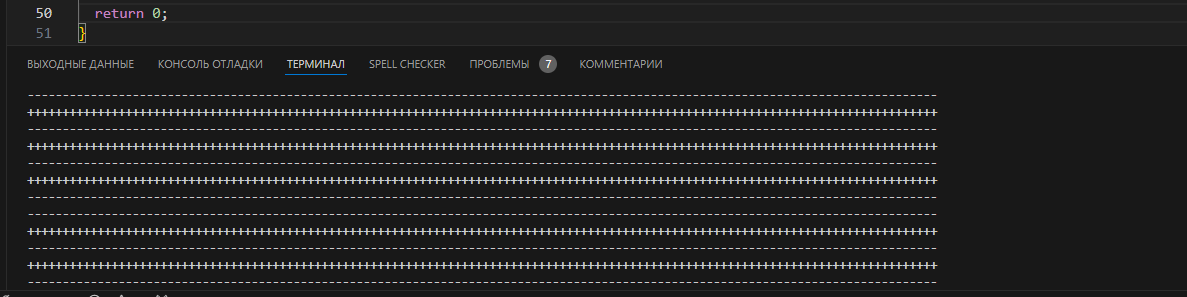
**5.** Теперь соберем программу несколько раз, подменяя в коде виды упорядочения памяти \_\_ATOMIC\_RELAXED,\_\_ATOMIC\_ACQUIRE /\_\_ATOMIC\_RELEASE,\_\_ATOMIC\_SEQ\_CST. Заметим, что при постановке режима \_\_ATOMIC\_RELAXED программа иногда выдает две одинаковые строки подряд (рис 5.1). 

Рис. 5.1 вывод программы с режимом \_\_ATOMIC\_RELAXED

**Вывод:** Мы познакомились с атомарными функциями и научились использовать для синхронизации потоков