ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

HỆ ĐIỀU HÀNH (CO2018)

LAB 3
SYNCHRONIZATION

GVHD: Nguyễn Quang Hùng

Sinh viên: Hoàng Tiến Hải - 2011152

Tp. Hồ Chí Minh, Tháng 2/2022

Mục lục

1 INTRODUCTION **2**

2 PRACTICE **2**

3 EXERCISES **2**

3.1 Problem1 2
3.2 Problem2 2
3.3 Problem 3 2

1 INTRODUCTION

2 PRACTICE

3 EXERCISES

3.1 Problem1

3.2 Problem2

Write a new program nosynch.c by copying the program cond_usg.c (Section 4) and then removing all entry and exit sections of the critical section solution.

1. What is your conclusion about the displayed outputs when executing these two programs nosynch.c and cond_usg.c.
2. What are **count_mutex** and **count_threshold_cv** used for?

SOLVE

- Chương trình nosynch.c bỏ các mutex và signal sẽ dẫn tới việc biến count sẽ được tăng một cách không đồng bộ giữa các thread, dễ dẫn tới trường hợp race condition. Trường hợp bỏ các mutex sẽ làm cho việc truy cập giá trị của biến count và thay đổi nó trên hai threads cùng lúc, dễ dẫn đến sai kết quả. Trường hợp bỏ condition signal sẽ làm cho việc thứ tự chương trình bị chạy sai hoặc chương trình rơi vào deadlock không thoát ra được do không có tín hiệu signal gửi tới thread 1 từ để giải phóng wait.
- Biến count_mutex hoạt động như một mutex lock. Sử dụng như một khoá bảo vệ dữ liệu tránh xảy ra race condition.
- Biến count_threshold_cv được dùng vào việc làm biến nhận vào cho hai wait và signal của condition, hai hàm này sẽ nhận và gửi tín hiệu giữa các thread với nhau và giải phóng tài nguyên khi nhận được tín hiệu.

3.3 Problem 3

In the Lab 2, we wrote a simple multi-thread program for calculating the value of pi using Monte-Carlo method. In this exercise, we also calculate pi using the same method but with a different implementation. We create a shared (global) count variable and let worker threads update on this variable in each of their iteration instead of on their own local count variable. To make sure the result is correct, remember to avoid race conditions on updates to the shared global variable by **using mutex locks and semaphore** (one version of program for each method). Compare the performance of this approach with the previous one in Lab 2.

SOLVE

– Với cách tính số pi theo phương pháp Monte Carlo như trong bài tập trước ở lab 2 đã làm. Chúng ta sẽ sử dụng lại sườn của phần trước với một vài sự thay đổi.

– Tiến hành thêm một biến đếm d để đếm trực tiếp các giá trị thoả mãn theo Monte Carlo của từng thread. Đồng thời khi thêm biến đếm toàn cục thì sẽ dẫn đến race conditions làm cho kết quả khi cập nhật biến đếm giữa các threads không được đồng bộ lẫn nhau. Vì vậy để tránh việc này xảy ra, ta sẽ sử dụng **mutex** để lock và unlock khi biến đếm có sự thay đổi.

```
static double d = 0;
pthread_mutex_t mutexlock = PTHREAD_MUTEX_INITIALIZER;
```

– Chúng ta sẽ sử dụng **pthread_mutex_lock()** mỗi khi có sự thay đổi biến đếm d để tránh race condition. Vì vậy ta sẽ thêm vào trước khi biến đếm được thay đổi,và sử dụng **pthread_mutex_unlock()** sau khi đã thay đổi xong giá trị của biến đếm.

```
for (long i = 0; i<lim_point; i++)
{
    double x_point = (double)rand_r(&seed)/RAND_MAX;
    double y_point = (double)rand_r(&seed)/RAND_MAX;
    if (x_point*x_point + y_point*y_point <= 1)
    {
        pthread_mutex_lock(&mutexlock);
        ++d;
        pthread_mutex_unlock(&mutexlock);
    }
}
```

– Các dữ kiện còn lại được giữ nguyên:

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

#define THREADS 8

struct data
{
    int seed;
    long point;
};

struct data thread_item[THREADS];

static double d = 0;
pthread_mutex_t mutexlock = PTHREAD_MUTEX_INITIALIZER;


void* in(void *arg)
{
    struct data *item;
    item = (struct data *) arg;
    long lim_point = item->point;
    int seed = item->seed;

    for (long i = 0; i<lim_point; i++)
    {
        double x_point = (double)rand_r(&seed)/RAND_MAX;
        double y_point = (double)rand_r(&seed)/RAND_MAX;
        if (x_point*x_point + y_point*y_point <= 1)
        {
            pthread_mutex_lock(&mutexlock);
            ++d;
            pthread_mutex_unlock(&mutexlock);
        }
    }

    pthread_exit(NULL);
}
```

```
int main()
{
    double nPoints;
    scanf("%lf", &nPoints);
    long point = nPoints / THREADS;

    pthread_t thread_id[THREADS];

    for (int i = 0; i<THREADS; i++)
    {
        time_t t_time;
        thread_item[i].seed = i + time(&t_time);
        thread_item[i].point = point;
        pthread_create(&thread_id[i], NULL, in, (void*)&thread_item[i]);
    }

    for (int i = 0; i<THREADS; i++)
    {
        pthread_join(thread_id[i], NULL);
    }

    double res = 4 * d / nPoints;
    printf("Calculate Pi is %lf\n", res);
    return 0;
}
```

– Đối với chương trình sử dụng **semaphore** để tránh race condition, ta sẽ sử dụng semaphore nhị phân với các khai báo:

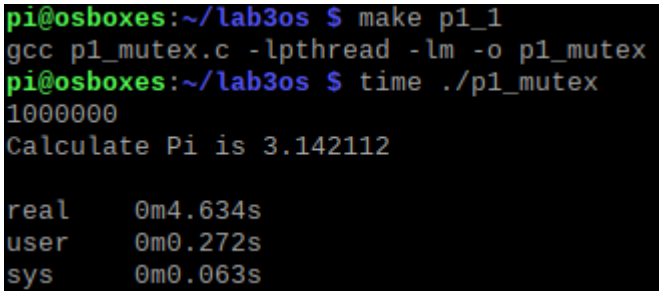
```
sem_init(&sem, 0, 1);
```

– Với việc sử dụng semaphore nhị phân sẽ tránh được race condition khi chỉ cho 1 thread được quyền thay đổi giá trị của biến đếm d và điều khiển thông qua **sem_wait()** và **sem_post()**.

```
for (long i = 0; i<lim_point; i++)
{
    double x_point = (double)rand_r(&seed)/RAND_MAX;
    double y_point = (double)rand_r(&seed)/RAND_MAX;
    if (x_point*x_point + y_point*y_point <= 1)
    {
        sem_wait(&sem);
        ++d;
        sem_post(&sem);
    }
}
```

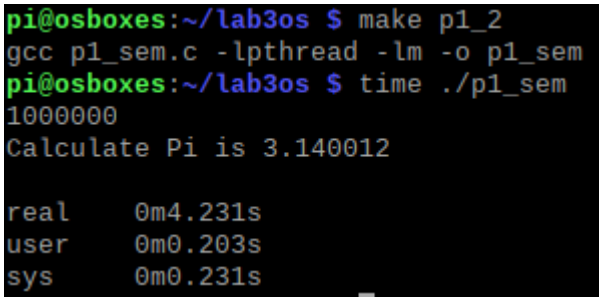
– Kiểm thử chương trình:

- Mutex program:



Hình 1

- Semaphore program:



Hình 2

– So sánh với phiên bản ở bài trước thì tốc độ chậm hơn do quá trình lock và unlock của mutex và semaphore tốn thêm thời gian chờ giữa các thread. Nhưng tránh việc race condition và nếu cải tiến có thể đạt được kết quả nhanh hơn bằng việc sử dụng các thuật toán.