UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

LABORATORIO MUTEX

Nifla Llallacachi, Manuel Angel

06 de Noviembre de 2024

0.1 Introducción

Este informe detalla las soluciones a los ejercicios 4.1, 4.3 y 4.5, extraídos de la sección de asignaciones de programación en Pthreads. Incluye explicaciones de las variables y funciones utilizadas, así como ejemplos de compilación.

0.2 SOLUCIONES

Ejercicio 4.1: Programa de histograma con Pthreads

Descripción

Este programa distribuye el cálculo de un histograma entre múltiples hilos, cada uno procesando un subconjunto de los datos generados aleatoriamente. El histograma se actualiza en una estructura compartida de manera segura utilizando un mutex.

Código en C++

Listing 1: Programa de histograma con mutexes

```
#include <iostream>
#include <pthread.h>
#include <vector>
#include <cstdlib>

int* HISTOGRAM;  // Memoria para ARRAY del Histograma
int N_intervalo;
int N_threads;
int N_datos;
double v_min;
double v_max;
std::vector<double> Vec_Datos;
pthread_mutex_t mutex;

void* CreandoHisto(void* rank) {
    long my_rank = (long)rank; // Identificador del thread
    int interva_local;
    double size_intervalo = (v_max - v_min) / N_intervalo;

    std::cout << "Hilo " << my_rank << " ha iniciado." << std::endl;

    for (int i = my_rank; i < N_datos; i += N_threads) {
        interva_local = (int)((Vec_Datos[i] - v_min) / size_intervalo);
        if (interva_local >= N_intervalo) {
            interva_local = N_intervalo - 1;
        }

        pthread_mutex_lock(&mutex); // Bloquea el mutex
```

```
        HISTOGRAM[interva_local]++;
        pthread_mutex_unlock(&mutex); // Desbloquea el mutex
    }

    std::cout << "Hilo " << my_rank << " ha terminado." << std::endl;
    return NULL;
}

int main(int argc, char* argv[]) {
    N_threads = strtol(argv[1], NULL, 10);
    N_datos = strtol(argv[2], NULL, 10);
    N_intervalo = strtol(argv[3], NULL, 10);
    v_min = atof(argv[4]);
    v_max = atof(argv[5]);

    HISTOGRAM = new int[N_intervalo]();
    pthread_t* thread_handles = new pthread_t[N_threads];
    pthread_mutex_init(&mutex, NULL); // Inicializa el mutex

    for (int i = 0; i < N_datos; i++) {
        Vec_Datos.push_back(v_min + ((double)rand() / RAND_MAX) * (v_max -
    }

    for (int thread = 0; thread < N_threads; thread++) {
        pthread_create(&thread_handles[thread], NULL, CreandoHisto, (void*)
    }

    for (int thread = 0; thread < N_threads; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }

    for (int i = 0; i < N_intervalo; i++) {
        std::cout << "Intervalo " << i << ": " << HISTOGRAM[i] << std::endl
    }

    pthread_mutex_destroy(&mutex); // Destruye el mutex
    delete[] thread_handles;
    delete[] HISTOGRAM;

    return 0;
}
```

Explicación

Variables

- **HISTOGRAM**: Array que almacena el conteo de valores en cada bin. Es compartido entre todos los hilos.

- **Vec_Datos**: Vector que contiene datos aleatorios para ser clasificados.
- **N_intervalo**: Número de bins (intervalos) en el histograma.
- **N_threads**:Número de hilos que se van a crear.
- **N_datos**:Número total de datos que se generarán para el histograma.
- **v_min**:Valor mínimo que definen el rango de los datos. Ayudan a dividir los datos en bins.
- **v_max**:Valor máximo que definen el rango de los datos. Ayudan a dividir los datos en bins.
- **mutex**:Mutex utilizado para proteger el acceso al array `HISTOGRAM` cuando varios hilos intentan actualizarlo al mismo tiempo.

Funciones

- **CreandoHisto**: Función de cada hilo que se encarga de asignar valores a bins. Cada hilo procesa una parte de data y calcula el bin correspondiente para cada valor. El mutex asegura que solo un hilo a la vez pueda incrementar los contadores de bins en `HISTOGRAM`.
- **pthread_mutex_lock y pthread_mutex_unlock**: Estas funciones bloquean y desbloquean el mutex, permitiendo que solo un hilo acceda y modifique `HISTOGRAM` a la vez.

Ejemplo de compilación y ejecución

```
$ g++ -o histogram 4.1.cpp -lpthread
$ ./histogram 4 1000 10 0.0 1.0
```

—

Ejercicio 4.3: Regla trapezoidal para la integración

Descripción

El programa divide la integral de una función en varios trapezoides y distribuye el cálculo entre los hilos. Cada hilo calcula una parte de la integral y suma su resultado en una variable global protegida por un mutex.

Código en C++

Listing 2: Regla trapezoidal con mutexes

```
#include <iostream>
#include <pthread.h>
#include <cmath>
```

```
double total_area = 0.0;
int N_threads;
double a, b;
int n;

pthread_mutex_t mutex;

double f(double x) { return x * x; }

void* Trapezoidal(void* rank) {
    long my_rank = (long)rank;
    int local_n = n / N_threads;
    double h = (b - a) / n;
    double local_a = a + my_rank * local_n * h;
    double local_b = local_a + local_n * h;

    double local_area = (f(local_a) + f(local_b)) / 2.0;
    for (int i = 1; i < local_n; i++) {
        double x = local_a + i * h;
        local_area += f(x);
    }
    local_area *= h;

    pthread_mutex_lock(&mutex);
    total_area += local_area;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main(int argc, char* argv[]) {
    N_threads = strtol(argv[1], NULL, 10);
    a = atof(argv[2]);
    b = atof(argv[3]);
    n = strtol(argv[4], NULL, 10);

    pthread_t* thread_handles = new pthread_t[N_threads];
    pthread_mutex_init(&mutex, NULL);

    for (int thread = 0; thread < N_threads; thread++) {
        pthread_create(&thread_handles[thread], NULL, Trapezoidal, (void*)t
    }

    for (int thread = 0; thread < N_threads; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }

    std::cout << " rea -total-=-" << total_area << std::endl;
```

```
    pthread_mutex_destroy(&mutex);
    delete[] thread_handles;

    return 0;
}
```

Explicación

Variables

- **total_area**: Variable compartida donde los hilos almacenan el área total calculada. **N_threads**:Número de hilos que se van a crear. **a y b**: Límites inferior y superior de la integral. **n**: Número de trapezoides en los que se divide el área bajo la curva. **mutex**: Mutex para controlar el acceso a `total_area`. **semaphore**: Semáforo opcional para proteger el acceso en la sección crítica (no es necesario en este caso, ya que solo usamos el mutex).

Funciones

- **f**: En este ejemplo, es una función cuadrática simple.
- **Trapezoidal**: Esta función calcula la contribución de cada hilo a la integral. Cada hilo calcula su parte de la integral y suma el área calculada en `total_area` usando un mutex para evitar condiciones de carrera.

Ejemplo de compilación y ejecución

```
$ g++ -o trapezoidal 4.3.cpp -lpthread
$ ./trapezoidal 4 0 10 1000
```

—

Ejercicio 4.5: Cola de tareas con Pthreads

Descripción

El programa implementa una cola de tareas en la que varios hilos consumidores procesan tareas conforme están disponibles, sincronizados mediante mutexes y variables de condición.

Código en C++

Listing 3: Cola de tareas con mutexes y condiciones

```
#include <iostream>
#include <pthread.h>
#include <queue>
#include <unistd.h>
```

```
std::queue<int> Cola;
bool final = false;
pthread_mutex_t cola_mutex;
pthread_cond_t condicion;

void* Funcion(void* R) {
    while (true) {
        pthread_mutex_lock(&cola_mutex);
        while (Cola.empty() && !final) {
            pthread_cond_wait(&condicion, &cola_mutex);
        }

        if (final && Cola.empty()) {
            pthread_mutex_unlock(&cola_mutex);
            break;
        }

        int Tarea = Cola.front();
        Cola.pop();
        pthread_mutex_unlock(&cola_mutex);

        std::cout << "-> Thread " << (long)R << " Procesando Tarea :" << T
        sleep(1);
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    pthread_t* thread_handles = new pthread_t[thread_count];
    pthread_mutex_init(&cola_mutex, NULL);
    pthread_cond_init(&condicion, NULL);

    for (int thread = 0; thread < thread_count; thread++) {
        pthread_create(&thread_handles[thread], NULL, Funcion, (void*)threa
    }

    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&cola_mutex);
        Cola.push(i);
        pthread_cond_signal(&condicion);
        pthread_mutex_unlock(&cola_mutex);
        sleep(1);
    }

    pthread_mutex_lock(&cola_mutex);
    final = true;
```

```
    pthread_cond_broadcast(&condicion);
    pthread_mutex_unlock(&cola_mutex);

    for (int thread = 0; thread < thread_count; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }

    pthread_mutex_destroy(&cola_mutex);
    pthread_cond_destroy(&condicion);
    delete[] thread_handles;

    return 0;
}
```

Explicación

Variables

- **Cola**: Cola donde se almacenan las tareas (números enteros). Es compartida entre todos los hilos.
- **final**: Booleano que indica si ya no se agregarán más tareas a la cola. Es compartida y ayuda a los hilos a saber cuándo terminar.
- **cola_mutex**: Mutex para proteger el acceso a `Cola` y `final`, evitando condiciones de carrera.
- **condicion**: Variable de condición para despertar a los hilos consumidores cuando hay una tarea disponible.

Funciones

- **Funcion**: Función que ejecutan los hilos consumidores. Cada hilo espera una tarea en la cola; si la cola está vacía, el hilo se suspende hasta que se le notifique que una tarea ha sido agregada.
- **pthread_mutex_lock y pthread_mutex_unlock**: Estas funciones bloquean y desbloquean el `cola_mutex`, permitiendo que solo un hilo acceda a `Cola` o `final`.

Ejemplo de compilación y ejecución

```
$ g++ -o task_queue 4.5.cpp -lpthread
$ ./task_queue 4
```

0.3 GitHub

En el siguiente link se encuentra el repositorio de Github:

https://github.com/AngelNifla/EPCC_COMPUTACION_PARALELA_Y_DISTRIBUIDA_2024/tree/master/LABS