Práctica 1: Una primera aproximación a la programación de sistemas concurrentes y distribuidos en C++

Programación de Sistemas Concurrentes y Distribuidos

Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivos

En esta práctica se empleará el lenguaje de programación C++ para trabajar con conceptos básicos de programación concurrente. En concreto los objetivos de esta práctica son:

- Crear y manipular programas concurrentes en C++.
- Implementar pequeños programas concurrentes que requieran cierta sincronización.
- Adquirir conocimiento empírico del funcionamiento de `threads` en C++ y su ejecución en el entorno de prácticas.

2. Varias versiones de un mismo programa

Vamos a ver variantes para ejecutar un programa concurrente en C++.

2.1. Primera aproximación

A continuación se muestra un programa concurrente en el que tres procesos análogos son creados y puestos en ejecución, hasta su terminación.

```
#include <iostream>
#include <thread>
#include <string>
#include <chrono>

using namespace std;

void saludo(string m, int retardo, int veces) {
```

```
    for(int i=1; i<=veces; i++) {
        cout << m+"\n";
        this_thread::sleep_for(chrono::milliseconds(retardo));
    }
}

int main() {
    thread th_1(&saludo, "Soy␣A␣", 100, 10),   //en marcha
           th_2(&saludo, "\t\tSoy␣B␣", 150, 15),
           th_3(&saludo, "\t\t\tSoy␣C␣", 300, 5);

    th_1.join();
    th_2.join();
    th_3.join();

    cout << "Fin\n";
    return 0;
}
```

2.2. Segunda aproximación

Cuando se tienen varios hilos que ejecutan el mismo código es más conveniente utilizar vectores de `threads`, como en el ejemplo siguiente.

```
#include <iostream>
#include <thread>
#include <string>
#include <chrono>

using namespace std;

void saludo(string m, int retardo, int veces) {
    for(int i=1; i<=veces; i++) {
        cout << m+"\n";
        this_thread::sleep_for(chrono::milliseconds(retardo));
    }
}

int main() {
    thread P[3];

    P[0] = thread(&saludo, "Soy␣1␣", 100, 10);
    P[1] = thread(&saludo, "\tSoy␣2␣", 150, 15),
    P[2] = thread(&saludo, "\t\tSoy␣3␣", 300, 5);

    P[0].join();
    P[1].join();
    P[2].join();

    cout << "Fin\n";
    return 0;
}
```

2.3. Tercera aproximación

Las aproximaciones anteriores son claras cuando el comportamiento de un proceso puede estar encapsulado en una única función. Sin embargo, es poco claro para el caso en que los procesos tengan comportamientos complejos, que requieran invocar varias funciones entre sí, con información propia compleja, etc.

Así, una aproximación más general consiste en encapsular en una clase de objetos el comportamiento de un proceso complejo, crear tantas instancias (objetos) de esa clase como sean necesarias, y lanzar la función deseada del objeto (cualquier de las suyas propias) en un `thread`, como se muestra a continuación

```
#include <iostream>
#include <thread>
#include <string>
#include <chrono>

using namespace std;

//-----------------------------------------------------
// Parte de Especificación: variables y funciones
// de los objetos de esta clase
// Normalmente iría en el fichero "Saludador.h"
// Dejamos todo público, para centrarnos en los conceptos
// de la concurrencia. Lo adecuado es distinguir entre
// parte pública y privada, con operadores para acceder
// a los atributos, por ejemplo. Lo estudiaréis en otras
// asignaturas

class Saludador {
public:
    Saludador(string mens,  //constructor suministrando datos
              int retardo,
              int veces);
    Saludador(); //constructor por defecto
    void run();
    //más funciones, si las hubiera
    string mens;
    int retardo, veces;
};
//-----------------------------------------------------
// Implementación (las funciones de los objetos de la clase)
// Normalmente iría en el fichero "Saludador.cpp"

Saludador::Saludador(string mens, int retardo, int veces) {
    this->mens = mens; //mens: parámetro de la función
                       //this->mens: variable propia
    this->retardo = retardo;
    this->veces = veces;
};

Saludador::Saludador() {
    mens = ""; //mens: parámetro de la función
               //this->mens: variable propia
```

```
    retardo = 0;
    veces = 0;
};

void Saludador::run() {
    for(int i=1; i<=veces; i++) {
        cout << mens +"\n";
        this_thread::sleep_for(chrono::milliseconds(retardo));
    }
};
//-------------------------------------------------------
int main() {

    //creación de los objetos: constructor con datos
    Saludador s1("Soy␣1", 100, 10),
              s2("\tSoy␣2", 150, 15),
              s3("\t\tSoy␣3", 10, 40);
    //creación de los objetos: constructor por defecto
    Saludador s4;

    //le damos datos a s4
    s4 = {"\t\t\tSoy␣4", 2, 12};
    //equivalente a
    // s4.mens = "\t\t\tSoy 4"; s4.retardo = 2; ...
    cout << "veces:␣" << s4.veces << endl;
    //puesta en ejecución
    thread th_1 = thread(&Saludador::run, s1);
    thread th_2 = thread(&Saludador::run, s2);
    thread th_3 = thread(&Saludador::run, s3);
    thread th_4 = thread(&Saludador::run, s4);
    //esperar a que vayan acabando
    th_1.join();
    th_2.join();
    th_3.join();
    th_4.join();

    return 0;
}
```

3. Ejercicios

3.1. Ejercicio 1

El ejercicio pide analizar qué hacen las distintas variantes mostradas en la sección anterior, y estudiar sus matices desde el punto de vista de la implementación. Para ello, hay que compilar y ejecutar cada uno de ellos. Tras varias ejecuciones, tratar de explicar el comportamiento del programa.

Debéis utilizar las fuentes habituales de documentación para la especificación de los elementos desconocidos, como

`this_thread::sleep_for(...)`, `chrono::milliseconds(...)`, etc.

3.2. Ejercicio 2

Se pide desarrollar una nueva versión del programa, siguiendo el esquema de la segunda aproximación mostrada, que se llamará `ejercicio_2.cpp`, en el que se lanzan 10 procesos, $1, 2, \ldots, 10$ de manera que el proceso i-ésimo se duerme durante un tiempo aleatorio de entre 100 y 300 milisegundos, y escribe el mensaje correspondiente ("Soy 1","Soy 2",...,"Soy 10") un número aleatorio de veces, entre 5 y 15. Para generar números aleatorios, se puede mirar documentación de las funciones `rand(), srand()` de la librería `stdlib`.

3.3. Ejercicio 3

Se trata de desarrollar una nueva versión del programa anterior, siguiendo un enfoque mixto de la segunda y tercera aproximaciones, separando el fichero de especificación `Saludador.h`, el fichero de implementación `Saludador.cpp` y el fichero con el programa principal `ejercicio_3.cpp`.

3.4. Ejercicio 4

Se trata de hacer un programa para el análisis de datos. Se pide escribir un programa concurrente de acuerdo a las siguientes especificaciones:

- Crea y rellena un vector de 100 reales, aleatorios
- Lanza tres procesos para analizar los datos: uno que calcula la media, otro el valor máximo y mínimo, y otro la desviación típica. Nótese que, aunque estos procesos comparten el vector de datos, no hay interferencias entre ellos, ya que únicamente acceden al vector para lectura, sin llevar a cabo ninguna modificación.
- Una vez que esos valores se han calculado, se informa de ellos por la salida estándar, de acuerdo al siguiente formato:

```
#datos: 100
media:  lo_que_sea
máx:    lo_que_sea
mín:    lo_que_sea
sigma:  lo_que_sea
```

El programa principal de este ejercicio se llamará `ejercicio_4.cpp`. También se deberán entregar los ficheros adicionales que sean necesarios para su compilación correcta.

4. Sobre la compilación

Para compilar un fuente `miProg.cpp`, que maneje threads, generando el ejecutable`miProg`, ejecutaremos

```
g++ -std=c++11 -pthread miProg.cpp -o miProg
```

En el caso de trabajar con un entorno de desarrollo (Codelite, QtCreator, etc.) deberemos añadir las opciones `-std=c++11 -pthread` en el lugar adecuado del entorno (buscar cómo añadir opciones al compilador y al "linker").

Por otro lado, es necesario tener presente que C++ incluye los `threads` desde la versión 11, que el compilador de GNU `g++` incorpora desde la versión 4.8. Por eso, si la versión de tu entorno en nuestros laboratorios es anterior (puedes ver la versión ejecutando desde una terminal `g++ --version`) deberás hacer lo siguiente:

- editar el fichero `.software` que tienes en tu "home"

  (por ejemplo, `gedit .software`)

- añadirle la línea

  ```
  gcc #Para trabajar con la última versión de g++
  ```

- cerrar la sesión actual y abrir una nueva

5. Entrega de la práctica

La práctica se realizará de forma individual. Cuando se finalice se debe entregar un fichero comprimido `practica1_miNIP.zip` (donde `miNIP` es el NIP del autor de los ejercicios) con el siguiente contenido:

1. Todos los ficheros con los fuentes solicitados
2. Un fichero de texto denominado `autor.txt` que contendrá el NIP, los apellidos y el nombre del autor de la práctica en las primeras líneas del fichero. Por ejemplo:

NIP	Apellidos	Nombre
345689	Rodríguez Quintela	Sabela

   También deberá contener:

   - una descripción de las principales dificultades encontradas para la realización de la práctica
   - una explicación del comportamiento observado en las ejecuciones del Ejercicio 1
   - para cada uno de los ejercicios 2, 3 y 4, el listado de los nombres de los ficheros fuente que conforman la solución solicitada así como la forma de compilarlos para obtener el ejecutable correspondiente.

Para la entrega del fichero `.zip` se utilizará el comando `someter` en la máquina `hendrix.cps.unizar.es`. Los alumnos pertenecientes a grupos de prácticas cuya primera sesión de prácticas se celebra el día 4 de octubre de 2017 deberán someter la práctica no más tarde del día 17 de octubre de 2017 a las 23:59. Los alumnos pertenecientes a grupos de prácticas cuya primera sesión de prácticas se celebra el día 18 de octubre de 2017 deberán someter la práctica no más tarde del día 31 de octubre de 2017 a las 23:59.

Hay que asegurarse de que la práctica funciona correctamente en los ordenadores del laboratorio (vigilar aspectos como los permisos de ejecución, juego de

caracteres utilizado en los ficheros, etc.). También es importante someter código limpio (donde se ha evitado introducir mensajes de depuración que no proporcionan información al usuario). El tratamiento de errores debe ser adecuado, de forma que si se producen debería informarse al usuario del tipo de error producido. Además se considerarán otros aspectos importantes como calidad del diseño del programa, adecuada documentación de los fuentes, correcto formateado de los fuentes, etc.

Para el adecuado formateado de los fuentes, es conveniente seguir unas pautas. Hay varias, y es posible que podáis configurar el entorno de desarrollo para cualquiera de ellas. Una posible, sencilla de seguir, es la "Google C++ Style Guide", que se puede encontrar en

`https://google.github.io/styleguide/cppguide.html`

Alternativamente, cualquiera que uséis en otras asignaturas de programación.