Práctica 1: Introducción a la programación concurrente 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 usará C++ para trabajar con conceptos básicos de programación concurrente. En concreto los objetivos de esta práctica son:

- Tener una primera aproximación a la implementación de programas concurrentes en C++.
- Adquirir experiencia en el manejo de *threads*.
- Tener constancia empírica de los efectos de la interacción entre procesos.

2. Trabajo previo

Como preparación para la sesión práctica es necesario:

- Repasar los aspectos básicos vistos en clase de teoría y problemas relativos a la sincronización de procesos mediante variables compartidas. También el primer ejemplo mostrado de cómo traducíamos el modelo de programa concurrente estudiado a su versión correspondiente en C++ (transparencias 20-22 de la lección 2).
- Buscar información relativa a la clase *string* de C++. Para el manejo de cadenas de caracteres, en primer curso habéis usado los "arrays" de caracteres, definidos como `char cad[LONG]`, junto con sus operaciones asociadas `strcpy, strcat, strlen, ...`. Podéis seguir usándolos. Sin embargo, en los enunciados y material que suministremos utilizaremos la clase `string` de C++. Se trata de una de las clases

estándares de C++. Suministra un conjunto muy extenso de operadores (más de 40), lo que permite un manejo muy flexible de esta clase de datos.

3. Introducción a los threads de C++

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

3.1. Primera aproximación

A continuación se muestra un programa concurrente en el que tres procesos análogos son creados para la ejecución de una misma función: `saludo`.

```
1  #include <iostream>
2  #include <thread>
3  #include <string>
4  #include <chrono>
5
6  using namespace std;
7
8  void saludo(string nombre, int retardo, int veces) {
9      for(int i=1; i<=veces; i++) {
10         // cout << "Soy " << nombre << endl;
11         cout << "Soy␣" + nombre + "\n";
12         //el thread que me ejecuta se bloquea durante "retardo" milisegundos
13         this_thread::sleep_for(chrono::milliseconds(retardo));
14     }
15 }
16
17 int main(int argc, char* argv[]) {
18     thread th_1(&saludo, "Aurora", 100, 10); //th_1 se pone en marcha
19     thread th_2(&saludo, "Baltasar", 150, 15);
20     thread th_3(&saludo, "Carmen", 300, 5);
21
22     th_1.join(); //me bloqueo hasta que "th_1" termine
23     th_2.join();
24     th_3.join();
25
26     cout << "Fin\n";
27     return 0;
28 }
```

Listado 1: Un primer ejemplo de programa con *threads*

3.2. Segunda aproximación

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

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

using namespace std;

void saludo(string nombre, int retardo, int veces) {
    for(int i=1; i<=veces; i++) {
        // cout << "Soy " << nombre << endl;
        cout << "Soy␣" + nombre + "\n";
        //el thread que me ejecuta se bloquea durante "retardo" milisegundos
        this_thread::sleep_for(chrono::milliseconds(retardo));
    }
}

int main(int argc, char* argv[]) {
    const int N = 3;
    thread P[N]; //de momento, ningún thread se pone en marcha

    P[0] = thread(&saludo, "Aurora", 100, 10); //P[0] se pone en marcha
    P[1] = thread(&saludo, "Baltasar", 150, 15),
    P[2] = thread(&saludo, "Carmen", 300, 5);

    P[0].join(); //me bloqueo hasta que "P[0]" termine
    P[1].join();
    P[2].join();
    // alternativamente, podíamos haber ejecutado
    //for (int i=0; i<N; i++) {
    // P[i].join();
    //}

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

Listado 2: Ejemplo de manejo de *arrays* de *threads*

4. Ejercicios

4.1. Ejercicio 1

El ejercicio pide analizar qué hacen las dos 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 los programas en los listados 1 y 2 (ficheros `practica_1_V1.cpp` y `practica_1_V2.cpp` del material suministrado, respectivamente) y, tras varias ejecuciones, tratar de explicar el comportamiento de los programas.

Debéis notar que en la línea 11 de ambos programas estamos usando el operador "+" de concatenación de strings: el resultado de `"Soy " + nombre + "\n"` en un string compuesto por la concatenación de `"Soy "`, del parámetro `nombre` y un salto de línea al final. En un programa secuencial, las instrucciones en las líneas 10 y 11 se comportarían de la misma manera. Sin embargo, ¿lo son en el caso de un programa concurrente? Para responder a esta pregunta podéis probar a ejecutar el programa con esas líneas comentadas alternativamente, y tratar de explicar las diferencias en el comportamiento.

Por otro lado, debéis utilizar las fuentes habituales de documentación para la especificación y uso de los elementos desconocidos, como

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

4.2. 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, 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`.

Para este ejercicio cambiamos la especificación del procedimiento anterior por el siguiente, donde `id` es el identificador del proceso que lo invoca (valor entre 1 y 10).

```
void saludo(int id, int retardo, int veces)
```

4.3. Ejercicio 3

4.3.1. Descripción del ejercicio

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

- Crea e inicializa 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 un tercero que calcula la desviación típica.
- Una vez que esos valores se han calculado, el programa principal 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 fichero con el programa correspondiente a este ejercicio se llamará `ejercicio_3.cpp`.

4.3.2. Análisis

Los procesos que intervienen ya se han especificado. Respecto a las interacciones entre ellos es preciso tener en cuenta las siguientes observaciones:

- Aunque los 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.
- Los procesos que calculan la media y el máximo y mínimo no pueden comenzar sus cálculos hasta que los datos hayan sido cargados.
- Para calcular la desviación es necesaria la media de los datos, por lo que el proceso que la calcule deberá esperar a que el que calcule la media termine
- El programa principal tampoco podrá escribir los resultados hasta que todos hayan sido calculados.

5. Sobre la compilación

Para compilar un fuente `miProg.cpp`, que maneje threads, generando el ejecutable `miProg`, ejecutaremos[1]

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

En el caso de trabajar con un entorno de desarrollo (Codelite, Codeblocks, 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 su estándar 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

[1] Las instrucciones de compilación dadas aquí corresponden al uso de las máquinas de los laboratorios del DIIS, tanto las máquinas CentOS como Unix. Para la instalación de un compilador adecuado en Windows 10, hay que seguir las instrucciones en el apartado *Indicaciones para la instalación del compilador de C++ de gnu en Windows 10* de la sección *Prácticas* del curso en moodle. Es importante leer el documento entero, y seguir los pasos indicados en la *ALTERNATIVA 2*.

6. Entrega de la práctica

Una vez terminada la práctica se debe entregar un fichero comprimido `practica_1_NIP.zip` (donde `NIP` es el NIP del alumno) con el siguiente contenido:

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

```
NIP: 345689
Apellidos: Rodríguez Quintela
Nombre: Sabela

Comentarios:
```

   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 los ejercicios 2 y 3, 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.

Generación del fichero `.zip` a entregar

Con el objetivo de homogeneizar los contenidos del fichero `.zip` vamos a proceder como sigue:

1. Creamos un directorio `practica_1_NIP` que contenga los ficheros que hay que entregar. Es importante tener presente que **se ha de hacer exactamente de esta manera**.[2]
2. Con el botón derecho del ratón sobre la carpeta seleccionamos la opción "Compress..." y le damos en nombre requerido, `practica_1_NIP.zip`
3. Alternativamente lo podemos hacer desde la terminal como sigue. Una vez creado el directorio `practica_1_NIP` con los ficheros pedidos ejecutamos lo siguiente desde la terminal:

```
zip -r practica_1_NIP.zip practica_1_NIP
```

[2]Como hay muchos alumnos matriculados, la gestión de las prácticas se hará de manera automática mediante *scripts*, que buscarán los ficheros con los nombres pedidos. Cualquier alteración o falta hará que no los puedan encontrar.

Con el fin de comprobar de que el `zip` contiene todos los ficheros que debe, y organizados adecuadamente, podéis ejecutar el script `pract_1_entrega_correcta.bash`. Leed la cabecera del fichero, que explica cómo utilizarlo.

Entrega del fichero en `hendrix`

Para la entrega del fichero `.zip` se utilizará el comando `someter` en la máquina `hendrix.cps.unizar.es`. Para ello, es necesario:

1) Pasar a vuestra cuenta de hendrix el fichero zip generado 2) Entrar en vuestra cuenta de hendrix 3) Desde ella, ejecutar lo siguiente[3]:

```
someter progc_21 practica_1_NIP.zip
```

Fechas de entrega de la práctica

La fecha de entrega depende de la fecha en que se haya tenido la sesión de prácticas:

- Las sesiones del 29/09 deben entregar no más tarde del 07/10, a las 20:00
- Las sesiones del 30/09 deben entregar no más tarde del 08/10, a las 20:00
- Las sesiones del 06/10 deben entregar no más tarde del 14/10, a las 20:00
- Las sesiones del 07/10 deben entregar no más tarde del 15/10, a las 20:00

Hay que asegurarse de que la práctica funciona correctamente en los ordenadores del laboratorio (hay que 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 de primero.

[3]Para que funcione, es necesario estar matriculado en la asignatura