Práctica 2: Resolución de problemas de sincronización mediante esperas activas

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 estudiará la resolución de problemas de sincronización (por condición y exclusión mutua) mediante esperas activas.

En concreto los objetivos de esta práctica son:

- Comprender y profundizar en la sincronización de procesos mediante esperas activas.
- Implementar sincronización mediante el uso de instrucciones de tipo `test-and-set`.
- Profundizar en el modelo de concurrencia de C++

2. Trabajo previo a la sesión en el laboratorio

Antes de la correspondiente sesión en el laboratorio, cada estudiante deberá leer el enunciado, analizar los problemas que en él se proponen y realizar un diseño previo de las soluciones sobre las que va a trabajar. *Los resultados de su trabajo de análisis y diseño los tendrá que expresar en un documento que presentará a los profesores antes del inicio de la sesión.* El documento debe contener como mínimo el nombre completo y el NIP del estudiante y, para cada ejercicio,

- la descripción de los datos compartidos, enumeración de los procesos que los comparten, e indicación de si requieren sincronización o no.

- un esbozo de alto nivel del código de los procesos indicando las zonas que están afectadas por la sincronización (ya sea zona de exclusión mutua o espera sincronizada).

El documento deberá llamarse `trabajo_previo_p2.pdf`, y su entrega es un pre-requisito para la realización y evaluación de la práctica.

3. Ejercicio 1

Considérese el entorno de datos siguiente, preparado para trabajar con vectores de datos de tipo *int*

```
1  const int N = 512;
2  const int N_BUSC = 8; //# de procesos buscadores
3  using VectInt = int[N] //"VectInt" es un "alias" para vector de int de dim. N
4
5  //-------------------------------------------------------
6  //Pre: 0 ≤ i ≤ d ≤ N − 1
7  //Post: result = Num  α ∈ [i, d].value = v[α]
8  void search(const VectInt v,
9              const int i, const int d,
10             const int value, int& result);
```

El ejercicio tiene como objetivo calcular el número de veces que aparece un valor en un vector mediante la ejecución de 9 procesos distintos: un proceso coordinador y 8 procesos buscadores. A continuación se detallan las acciones que realiza cada proceso:

- El proceso coordinador:
  1. Carga los datos del vector desde el fichero “datos.txt” que se suministra con el material adicional de la práctica
  2. “Avisa” a los procesos buscadores de que los datos están cargados
  3. Espera a que cada proceso buscador termine su trabajo y le “avise” de ello
  4. Muestra por la salida estándar el número total de veces que aparece el valor buscado en el vector
- Por su parte, cada proceso buscador:
  1. Espera a que el proceso coordinador haya cargado los datos
  2. Lleva a cabo su trabajo, calculando el número de veces que aparece el valor indicado en un trozo del vector
  3. “Avisa” al coordinador de que ha finalizado su trabajo, y termina
- En cuanto al programa principal, este solicita al usuario que introduzca el valor a buscar, un entero entre 1 y 25, por la entrada estándar (teclado), lanza los 9 procesos indicados con los parámetros adecuados y espera a que estos terminen.

4. Análisis

A raíz del enunciado, antes de diseñar la solución deberíamos plantearnos y considerar unas cuantas cuestiones importantes relacionadas con la sincronización entre los procesos involucrados:

1. ¿Qué parte del vector debería resolver cada uno de los procesos buscadores?
2. ¿Cómo puede el coordinador avisar a los buscadores de que ha cargado los datos y cómo pueden estos esperar hasta que esto ocurra?
3. Como cada buscador va a trabajar sobre un trozo distinto del vector (y, además, el vector de datos solo lo usan para leer datos), no hay problemas de interferencias entre ellos. Sin embargo, ¿cómo se puede gestionar que cada buscador obtiene el número de ocurrencias del valor en un trozo distinto?
4. ¿Cómo va cada buscador a avisar al coordinador de que ha hecho su trabajo?
5. ¿Cómo espera el coordinador a que todos los buscadores hayan acabado?

El fichero fuente que contenga el procedimiento `main` de este ejercicio deberá llamarse `main_p2_e1.cpp`.

5. Ejercicio 2

Se trata de hacer una segunda versión del programa anterior en el que simplemente queremos saber en qué trozo del vector hay un mayor número de ocurrencias del valor introducido por el usuario. El comportamiento de los procesos es similar al del ejercicio anterior, pero además los buscadores deberán ser capaces de actualizar correctamente los parámetros `maxVeces`, `indMin` e `indMax` cuando sea necesario. Estos parámetros representan el mayor número de ocurrencias encontradas hasta ese instante y los índices menor y mayor del trozo del vector donde se encontró ese máximo, respectivamente. Dado que todos lo procesos han de acceder concurrentemente a estos tres parámetros, se le suministra también una variable, por referencia, del tipo `std::atomic_flag`, que permite utilizar la instrucción `test-and-set` para asegurar el acceso en exclusión mutua, tal y como se ha mostrado en clase[1].

```
1 // Complétese la especificación de acuerdo a lo dicho más arriba
2 void search(const VectInt v,
3             const int i, const int d, const int value,
4             int& maxVeces, int& indMin, int& indMax,
5             atomic_flag& tas);
```

El fichero fuente que contenga el procedimiento `main` de este ejercicio deberá llamarse `main_p2_e2.cpp`.

[1] Véase `http://www.cplusplus.com/reference/atomic/atomic_flag/test_and_set/` para su especificación y ejemplo de uso.

6. Entrega de la práctica

Cuando la práctica se finalice, el estudiante debe entregar un fichero comprimido `practica_2_NIP.zip` (donde `NIP` es el NIP del autor de la solución) con el siguiente contenido:

1. Los ficheros fuente programados (con los nombres concretos que se indican en las instrucciones)
2. Los ficheros `Makefile_p2_e1` y `Makefile_p2_e2` que compilen los ejercicios 1 y 2, respectivamente, generando los ejecutables `main_p2_e1` y `main_p2_e2`.

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_2_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_2_NIP.zip`
3. Alternativamente lo podemos hacer desde la terminal como sigue. Una vez creado el directorio `practica_2_NIP` con los ficheros pedidos ejecutamos lo siguiente desde la terminal:

```
zip -r practica_2_NIP.zip practica_2_NIP
```

Con el fin de comprobar que el `zip` contiene todos los ficheros que debe, y organizados adecuadamente, podéis ejecutar el script `pract_2_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`

```
someter progc_21 practica_2_NIP.zip
```

[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.

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 14/10 deben entregar no más tarde del 22/10, a las 20:00
- Las sesiones del 20/10 deben entregar no más tarde del 28/10, a las 20:00
- Las sesiones del 21/10 deben entregar no más tarde del 29/10, a las 20:00
- Las sesiones del 27/10 deben entregar no más tarde del 04/11, 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.