Práctica 3: Programación con semáforos 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 estudiará la resolución de problemas de sincronización mediante semáforos. En concreto, los objetivos de esta práctica son:

- comprender y profundizar en la sincronización de procesos,
- resolver problemas de sincronización de procesos utilizando semáforos,
- y 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 entregará en papel a los profesores antes del inicio de la sesión (en el mismo laboratorio). 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 identificación de los semáforos necesarios para sincronizar la actividad de estos procesos.
- un esbozo de alto nivel del código de los procesos indicando las zonas que están afectadas por la sincronización.

La entrega de este documento es un prerrequisito para la realización y evaluación de la práctica. El documento debe estar correctamente escrito, sin faltas de ortografía, y cumplir los mínimos exigibles de presentación. En caso contrario, no se valorará.

3. Semáforos en C++

C++ no tiene datos de tipo semáforo con la semántica vista en clase. Por este motivo se suministra la clase `Semaphore_V4`, correspondiente a los ficheros `Semaphore_V4.hpp` y `Semaphore_V4.cpp`. Se proporciona también un ejemplo de uso `pruebaSemaforos.cpp`.

Si leéis detenidamente la especificación de la clase implementada veréis que contiene dos constructores diferentes:

- `Semaphore(const int n, const string info = ‘‘’’)`

  Crea un dato de tipo semáforo e inicializa su valor a n (número de permisos). Por tanto, corresponde con la versión vista en clase.

- `Semaphore(const string info = ‘‘’’)`

  Requiere una posterior inicialización mediante `void setInitValue(const int n)` como paso previo a su uso (el estado del semáforo después de la secuencia `Semaphore s; s.setInitValue(2);` es el mismo que `Semaphore s(2);`). Este segundo constructor resulta de utilidad cuando se declara un *array* de semáforos (ya que, por ejemplo, diferentes semáforos del *array* pueden requerir diferentes números de permisos).

Los dos constructores anteriores tienen un parámetro, optativo, `String info`. En caso de que no se suministre en el constructor, tomará como valor el string vacío. Esta información se usará en caso de que se compile el programa con la opción de generar un fichero de log, que almacena eventos del programa generando una traza de la ejecución del programa. Este fichero de log es útil para posteriormente analizar el comportamiento del sistema implementado y verificar su correcto funcionamiento, conforme se ha explicado en clase. El parámetro `info` sirve para asignar un nombre al semáforo que se está declarando. Posteriormente, durante la ejecución del sistema, este nombre se añadirá a todos los eventos que grabe el `Logger` en la traza de ejecución, permitiendo diferenciar de esta manera qué eventos corresponden a cada semáforo particular (si `info` es el string vacío, el `Logger` no generará eventos). En el `Anexo I` se explica en detalle el formato y contenido de estos ficheros de log. Esta aproximación se puede utilizar solo durante la etapa de desarrollo, o bien dejarla de manera permanente. Nótese que desde el punto de vista del código, la única diferencia radica en el momento de la declaración de la variable (ejecución del constructor). Las sucesivas operaciones se denominan de la misma manera.

La especificación de la clase semáforo también ofrece dos versiones de las operaciones `wait` y `signal`, donde la única diferencia radica en el número de permisos que están involucrados en la acción. La versión sin parámetros corresponde con la explicada en clase (el valor del semáforo se incrementa/decrementa en una unidad respetando las reglas semánticas de la primitiva de sincronización). La semántica de la segunda versión, una generalización de la anterior, se explica en el fichero `Semaphore_V4.hpp`.

Una forma de generar programas con y sin la opción de generar el log es la que se muestra en el ejemplo `pruebaSemaforos.cpp`, mediante la compilación condicional definida por `#ifdef LOGGING_MODE ... #else ... #endif` en el main, junto con `-D LOGGING_MODE` en la invocación al compilador en el Makefile correspondiente, que se encarga de definir `LOGGING_MODE`. Esta combinación hace que

al compilar el programa y generar el ejecutable, este se genere como si el fuente tuviera o bien las instrucciones en el caso del `#if` o del `#else`.

4. Trabajo a realizar en la práctica

La clase `BoundedQueue` implementa una estructura de datos genérica de tipo cola FIFO con capacidad limitada (la capacidad se debe establecer en el momento de su declaración). La especificación de la clase declara un constructor y un conjunto de operaciones que permiten gestionar el uso habitual de una de estas estructuras de datos (análogas a las que habéis estudiado en la asignatura de EDA). La semántica concreta de estas operaciones se puede consultar en el fichero `BoundedQueue.hpp` proporcionado como parte del material de la práctica (también se suministra la implementación de la clase en el fichero `BoundedQueue.cpp`).

El objetivo de este ejercicio es programar un sistema concurrente formado por:

- 4 procesos *escritores*: Cada escritor inserta 8 mensajes de texto en la cola. El contenido de estos mensajes está cifrado conforme a la regla de sustitución definida en el *cifrado César*[1]. El Anexo II contiene la lista de mensajes a utilizar. Todos los procesos cifradores usarán esta misma lista (que deberá estar declarada dentro de la función que los threads de cifrado ejecuten).
- 5 procesos *lectores*: Cada lector saca 6 mensajes de la cola. Después de descifrar su contenido muestra por la salida estándar su identificador de proceso y el texto obtenido.

La cola es una estructura compartida que tiene una capacidad máxima de 10 mensajes de texto. Dado que las operaciones de la clase `BoundedQueue` no gestionan el acceso concurrente de múltiples procesos, es necesario considerar este aspecto, tal y como se describe a continuación.

La primera parte de esta práctica consiste en desarrollar una versión de la cola FIFO que garantice un acceso concurrente correcto. Para ello hay que desarrollar una nueva clase, denominada `ConcurrentBoundedQueue`, acorde a la especificación en el fichero `ConcurrentBoundedQueue.hpp`. Es necesario completar el fichero `ConcurrentBoundedQueue.cpp` que se entrega con el código de cada una de las funciones propuestas. La implementación debe llevarse a cabo con la técnica de paso de testigo mostrada en clases de teoría (lección 6, páginas 6-7).

La segunda parte del ejercicio consiste en programar el sistema concurrente descrito al comienzo de esta sección. Para ello, hay que completar el código en el fichero `main_p3_e1.cpp`, de manera que la ejecución de la invocación `main_p3_e1` se comporte conforme a las especificaciones dadas.

[1] https://es.wikipedia.org/wiki/Cifrado_César

5. Material proporcionado para resolver el ejercicio

En la Web de la asignatura está disponible un fichero `practica3_NIP.zip` que contiene los ficheros que hay que entregar, adecuadamente organizados. También hay un *script* (`pract3_correcta.bash`) que prueba la correcta organización del zip entregado. Un error detectado por este *script* indicaría un problema en lo entregado, que generará una calificación de suspenso en la práctica.

Recuérdese que debemos modificar el contenido de los ficheros `ConcurrentBoundedQueue.cpp` y `main_p3_e1.cpp`, y entregar un documento `informe_p3.pdf` conforme a las pautas especificadas en la sección de entrega. Este informe debe añadirse al directorio raíz del fichero `practica3_NIP.zip`.

6. Entrega de la práctica

La práctica se realizará de forma individual. Cuando se finalice se debe entregar un fichero comprimido `practica3_miNIP.zip` (donde `miNIP` es el NIP del autor de los ejercicios) con el siguiente contenido:

1. Los ficheros proporcionados, tanto los que hay que modificar como los que no, como aquellos ficheros adicionales que el alumno considere necesario. Si esto último ocurriera, el alumno deberá adaptar el fichero `Makefile_p3_e1` para que la compilación se lleve a cabo correctamente.
2. Un documento en formato PDF, denominado `informe_p3.pdf` que debe contener la siguiente información:
   - el nombre completo del alumno y su NIP
   - una breve explicación de las decisiones de diseño adoptadas en su solución
   - una descripción de las principales dificultades encontradas para la realización de la práctica
   - una explicación del comportamiento observado en las ejecuciones del ejercicio
   - para cada uno de los ejercicios, el listado de los nombres de los ficheros fuente que conforman la solución solicitada.

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 tercera sesión de prácticas se celebra el día 30 de octubre de 2019 deberán someter la práctica no más tarde del día 13 de noviembre de 2019 a las 23:59. Los alumnos pertenecientes a grupos de prácticas cuya tercera sesión de prácticas se celebra el día 6 de noviembre de 2019 deberán someter la práctica no más tarde del día 20 de noviembre de 2019 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.

Anexo I

Supongamos que en el programa principal se han declarado dos semáforos con valor inicial 1 y nombres "s1" y "s2", respectivamente. Supongamos también que se ha compilado con la opción `-D LOGGING_MODE`, para generar un fichero de log con la historia correspondiente a la ejecución del programa.

```
Semaphore s1(1, "s1"),
          s2(1, "s2");
```

A continuación se muestra un fragmento del fichero de log del ejemplo correspondiente a una ejecución del programa anterior. Este fichero es similar al que se generará en esta práctica. La primera línea contiene la cabecera del log (se genera automáticamente), donde se especifica el formato y los campos de información almacenados en el fichero. Estos campos son los siguientes: el identificador del proceso invocador, el nombre del semáforo (información suministrada en su declaración) sobre el que se invoca la operación, el tipo de evento de interés (WAIT o SIGNAL sobre un semáforo), el valor natural asociado al semáforo, el *time-stamp* (instate de reloj de ejecución del evento) y un *ticket* interno que se utiliza para garantizar la escritura ordenada de los mensajes en el fichero log compartido.

```
threadID,sectionID,event,val,ts,ticket
...
id_140640893953856,s1,WAIT,0,1571314605770104130,61
id_140640893953856,s2,SIGNAL,1,1571314605770118427,62
id_140640893953856,s1,SIGNAL,3,1571314605770118774,63
id_140640893953856,s2,WAIT,0,1571314605770129745,64
id_140640893953856,s1,WAIT,2,1571314605770131553,65
id_140640893953856,s1,SIGNAL,3,1571314605770135992,66
id_140640893953856,s2,SIGNAL,1,1571314605770142420,67
...
```

También se muestran las líneas correspondientes a ocho eventos de ejecución. Estos eventos corresponden a invocaciones a los semáforos previamente declarados por parte de los procesos del sistema concurrente.

Anexo II

```
const string mensajes[] = {
  "a␣leopard␣never␣changes␣its␣spots",
  "a␣friend␣in␣need␣is␣a␣friend␣indeed",
  "kill␣two␣birds␣with␣one␣stone",
  "every␣cloud␣has␣a␣silver␣lining",
  "things␣often␣happen␣when␣you␣least␣expect␣them␣to",
  "he␣who␣laughs␣last␣laughs␣longest",
  "a␣cat␣in␣gloves␣catches␣no␣mice",
  "actions␣speak␣louder␣than␣words"
};
```