Práctica 3: Resolución de problemas de sincronización mediante semáforos

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 pareja de estudiantes 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 y presentará a los profesores antes del inicio de la sesión.* El documento debe contener como mínimo el nombre completo y el NIP de los dos estudiantes y, para cada ejercicio,

- un diseño de alto nivel de la solución que incluya 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.

El documento deberá llamarse `informe_P3_NIP1_NIP2.pdf` (donde `NIP1` **es el NIP menor** y `NIP2` **es el NIP mayor** de la pareja), y deberá entregarse antes del comienzo de la sesión de prácticas utilizando el comando `someter` en la máquina `hendrix.cps.unizar.es`

```
someter progc_21 informe_P3_NIP1_NIP2.pdf
```

Su entrega es un pre-requisito para la realización y evaluación de la práctica.

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`.

La clase implementada tiene un único constructor `Semaphore(const int n)`. El parámetro corresponde al número de permisos que se le asocia inicialmente, equivalente al usado en la notación algorítmica utilizada en clases de teoría.

La especificación de la clase semáforo ofrece dos versiones de `wait` y dos de `signal`. La diferencia entre ellas radica en el número de permisos que están involucrados en la acción. La versión sin parámetros (`sem.signal(), sem.wait()`) se 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 (`sem.signal(3), sem.wait(2)`, por ejemplo), es una generalización de la anterior, que se explica en los comentarios del fichero `Semaphore_V4.hpp`.

4. Ejercicio a desarrollar

Un locutorio telefónico dispone de `N_CAB` cabinas de uso individual. Los clientes (hay `N_USER` clientes) entran al locutorio cuando alguna de las cabinas está libre y no está puesto el cartel de *limpiando*, eligen a continuación una concreta de las libres, usan el teléfono durante un tiempo (aleatorio) y luego abandonan el local por la puerta de salida.

Por otro lado, cada cierto tiempo (`PER_CLEANING`) el propietario va a limpiar todas las cabinas. Para ello, pone un cartel indicando el cierre temporal del locutorio, espera a que las `N_CAB` cabinas estén libres, limpia las cabinas y retira el cartel, reanudando el servicio normal.

El desarrollo de la práctica se llevará a cabo como sigue:

- Plantéese un diseño de la solución acorde con el esquema mostrado en el Anexo-I, utilizando la misma declaración de constantes y procedimientos correspondientes a los procesos involucrados.

- Es necesario que el programa desarrollado incluya la generación de un fichero de log, denominado `_log_.log`, de acuerdo con las notas en el Anexo-I (las siguientes secciones explican cómo se genera el fichero de log). Las instrucciones concretas para las opciones al compilador pueden verse en el fichero fuente `pruebaSemaforos.cpp` (líneas 20-28) y el `Makefile` (línea 15) que le acompaña en el directorio `Semaphore_V4`.

El Anexo-II muestra un ejemplo de log (parcial) posible durante la ejecución del programa solicitado.

5. Generación de ficheros de log

En muchas situaciones es interesante almacenar información sobre la historia de ejecución, con el fin de poder analizar propiedades comportamentales, detectar problemas e inconsistencias, etc. En el caso de un programa secuencial es sencillo: se escriben eventos en un fichero conforme van sucediendo. En el caso de un programa concurrente, por cuestiones de entrelazados, no es tan sencillo, ya que puede que los eventos no se guarden en el orden en que ocurrieron. Para ello suministramos la librería `logger_V3`. La generación del log se puede usar solo durante la etapa de desarrollo, para poner a punto el programa, pero también se puede dejar para la etapa de explotación, pues podría ser útil para detectar posibles causas de problemas durante su explotación. En cualquier caso, que el programa genere un fichero de log o no se puede activar/desactivar en la compilación del mismo.

Una forma de compilar 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. Así, si en la invocación al compilador en el makefile aparece la opción `-D LOGGING_MODE`, entonces se define la variable interna `LOGGING_MODE`. Si no aparece, no está definida. Por otro lado, las líneas 20 a 27 del fichero `pruebaSemaforos.cpp` analizan si dicha variable está definida. Si no lo está, se ejecutará la línea 26, con lo que `ADD_EVENT(e)` será la instrucción vacía. En este caso, las líneas que contengan un `ADD_EVENT` están vacías.

En caso contrario se ejecutarán las líneas 21 (incluye las definiciones del logger), la 22 (generando el fichero de log `_log_.log`) con un buffer de capacidad 1024 (lo eventos se van guardando en una estructura de datos intermedia y son pasados al fichero cuando se hayan generado 1024 eventos o cuando se acabe el programa), y haciendo que la instrucción `ADD_EVENT` se defina como la línea 25 establece. Ahora la línea 96, por ejemplo, ya no será vacía, sino que será, exactamente, `{_logger.addMessage(nada+e);}`. Esta instrucción almacena un evento en el fichero de log. El evento se forma a partir del parámetro `e` (un string) junto con información del sistema (identificador del thread que lo ha generado, el timestamp del momento en que ha sucedido, ...). El contenido de `e` dependerá del programa concreto, que es lo que le dará el significado.

6. Análisis de ficheros log

Cada ejecución del programa genera un fichero log que contiene la traza de eventos, conforme a lo explicado en la sección anterior. Este log se puede analizar posteriormente para chequear si el programa se comporta conforme a las especificaciones. Ese chequeo se realizará utilizando una herramienta de análisis disponible en

`https://mizar.unizar.es/pscd`

La herramienta está configurada para evaluar un conjunto de fórmulas de lógica temporal que garanticen ciertas propiedades de corrección del programa. El uso de la herramienta es muy simple (sólo es necesario seleccionar la práctica concreta y "subir" el fichero log que se desea analizar). No obstante, dispone de un manual de uso.

7. Entrega de la práctica

Una vez la práctica terminada, los dos componentes de la pareja deben entregar, cada uno desde su cuenta, el mismo fichero comprimido `practica_3_NIP1_NIP2.zip` (donde `NIP1` **es el NIP menor** y `NIP2` **es el NIP mayor** de la pareja) con el siguiente contenido:

1. El fichero `practica_3.cpp` con el main de programa
2. El directorio `librerias` que se suministra (que, a su vez, contiene las librerías de semáforos y para generar ficheros de log)
3. El fichero `Makefile_p3` que compila el fuente, generando el ejecutable `practica_3`
4. Todos los demás ficheros requeridos para que la ejecución de `make -f Makefile_p3` genere el ejecutable pedido

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_3_NIP1_NIP2` que contenga los ficheros que hay que entregar. Es importante tener presente que **se ha de hacer exactamente de esta manera**.
2. Con el botón derecho del ratón sobre la carpeta seleccionamos la opción "Compress..." y le damos en nombre requerido, `practica_3_NIP1_NIP2.zip`
3. Alternativamente lo podemos hacer desde la terminal como sigue. Una vez creado el directorio `practica_3_NIP1_NIP2` con los ficheros pedidos ejecutamos lo siguiente desde la terminal:

```
zip -r practica_3_NIP1_NIP2.zip practica_3_NIP1_NIP2
```

Con el fin de comprobar que el `zip` contiene todos los ficheros que debe, y organizados adecuadamente, podéis ejecutar el script `pract_3_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_3_NIP1_NIP2.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 28/10 deben entregar no más tarde del 7/11, a las 20:00
- Las sesiones del 3/11 deben entregar no más tarde del 13/11, a las 20:00
- Las sesiones del 10/11 deben entregar no más tarde del 20/11, a las 20:00
- Las sesiones del 11/11 deben entregar no más tarde del 21/11, 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.

Anexo-I

Esbozo del código de los procesos

```
1  const int N_USER = 20;   //num de usuarios
2  const int N_TIMES_USER = 30; //num de veces un usuario utiliza el sistema
3  const int N_TIMES_CLEANING = 5; //num de limpiezas a ejecutar
4  const int PER_CLEANING = 100;  //tiempo de espera antes de cada limpieza
5  const int N_CAB = 4; //num de cabinas en el locutorio
6
7  Process usuario(i: 1..N_USER ...)::
8     ADD_EVENT("USER_BEGIN_" + to_string(i));
9     for j:=1..N_TIMES_USER
10          ADD_EVENT("USER_WAIT_" + to_string(i) + "," + to_string(j));
11       //espera a que no se esté limpiando y haya alguna cabina disponible
12       //entra en el locutorio
13       //entra en una cabina de entre las libres. Supongamos es la k (0 <= k < 4)
14          ADD_EVENT("USER_IN_" + to_string(i) + "," + to_string(j) + "," +
                ↪ to_string(k));
15       //usa la cabina durante un tiempo aleatorio [10,40]
16          ADD_EVENT("USER_OUT_" + to_string(i) + "," + to_string(j) + "," +
                ↪ to_string(k));
17       //tiempo aleatorio de espera, [20,50], antes de volver a usa el locutorio
18    end
19    ADD_EVENT("USER_END_" + to_string(i));
20 end
21
22 Process cleaning(...)::
23    ADD_EVENT("CLEANING_BEGIN");
24    for j:=1..N_TIMES_CLEANING
25       //espera PER_CLEANING
26          ADD_EVENT("CLEANING_WAIT" + "," + to_string(j));
27       //marca que va a entrar y espera se liberen las cabinas
28          ADD_EVENT("CLEANING_IN" + "," + to_string(j));
29       //tiempo aleatorio, [80,120], simulando ejecución de la limpieza
30          ADD_EVENT("CLEANING_OUT" + "," + to_string(j));
31    end
32    ADD_EVENT("CLEANING_END");
33 end
```

Por otra parte, `main` generará los eventos siguientes justo al principio y al final, respectivamente,

```
1  ADD_EVENT("MAIN_BEGIN");
2  ...
3  ADD_EVENT("MAIN_END");
```

Anexo-II

Ejemplo de log parcial para el programa solicitado.

```
ID,ts,threadID,ticket,event,a1,a2,a3,...
id_139874549184320,1634567931851068300,139874549184320,1,BEGIN_MAIN
id_139874549184320,1634567931851279601,139874549180160,2,CLEANING_BEGIN
id_139874549184320,1634567931851517419,139874540787456,3,USER_BEGIN_0
id_139874549184320,1634567931851658495,139874549180160,4,CLEANING_WAIT,0
id_139874549184320,1634567931851688662,139874532394752,5,USER_BEGIN_1
...
id_139874549184320,1634567931968453972,139873654662912,59,USER_WAIT_16,1
id_139874549184320,1634567931975443603,139873629484800,60,USER_OUT_19,0,3
id_139874549184320,1634567931975468747,139873629484800,61,USER_WAIT_19,1
id_139874549184320,1634567931975508956,139874132817664,62,USER_IN_8,0,3
id_139874549184320,1634567931988559152,139874515609344,63,USER_OUT_3,0,1
id_139874549184320,1634567931988574780,139873637877504,64,USER_IN_18,0,1
...
id_139874549184320,1634567936150434182,139874107639552,1857,USER_OUT_11,29,0
id_139874549184320,1634567936150456993,139874107639552,1858,USER_END_11
id_139874549184320,1634567936150655349,139874549184320,1859,END_MAIN
```