Práctica 4: Programación con monitores 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 monitores. En concreto, los objetivos de esta práctica son:

- comprender y profundizar en la sincronización de procesos,
- resolver problemas de sincronización de procesos utilizando monitores,
- 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 enumeración de los procesos involucrados y una descripción completa del monitor necesario 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_P4_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_P4_NIP1_NIP2.pdf
```

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

3. Ejercicio a desarrollar

En esta práctica se va a resolver el mismo problema que en la práctica 3, pero utilizando monitores como medio para la sincronización de los procesos. El Anexo-I muestra el esqueleto de los procesos y del monitor. El monitor encapsula los aspectos de sincronización. Vemos que, justo antes de terminar la ejecución de cualquiera de las cuatro operaciones del monitor, se añade un evento, que se corresponde con alguno de los eventos generados en la práctica anterior.

Respecto al enunciado de la práctica 3 vamos a añadir un nuevo requisito: es necesario dar prioridad al servicio de limpieza.

4. Generación de ficheros de log

En la práctica anterior ya hemos visto cómo se podía añadir la generación de un fichero de log. Recuérdese que lo hacíamos mediante la inclusión en el programa principal de la macro `#ifdef LOGGING_MODE ... #else ... #endif`. Ahora la situación es un poco distinta: el fichero de log, además de en el programa principal, lo necesitamos en el monitor, que se encuentra especificado en el fichero `ControlCabinas.hpp` e implementado en el fichero `ControlCabinas.cpp`, tal y como se especifica en la sección 6. El `logger` definido en el programa principal va a ser usado también por el monitor. Para ello, hemos de añadir en `ControlCabinas.hpp` las siguientes líneas

```
#ifdef LOGGING_MODE
    #include <Logger.hpp>
    extern Logger _logger; //_logger está definido en algún otro fichero
    extern const string nada;//nada está definida en algún otro fichero
    #define ADD_EVENT(e) {_logger.addMessage(nada+e);} //generar evento
#else
    #define ADD_EVENT(e) // nada
#endif
```

Con ellas se está diciendo que se va a usar una variable `Logger _logger` que *ya ha sido declarada en alguna otra parte* (calificador `extern`). Esta es una manera de poder compartir variables entre distintos módulos de un programa. Es una técnica que se ha de usar solo en casos en que opciones alternativas no sean más recomendables. Por ejemplo, aquí podríamos haber puesto en las operaciones del monitor un parámetro `Logger _logger`, que se pasase desde las invocaciones del programa principal, donde se declara la variable. Sin embargo, dado que no siempre vamos a querer generar el fichero

de log (recuérdese que es habitual que se use durante las etapas del desarrollo de la aplicación, pero no durante su explotación) la adición de ese parámetro *desnaturalizaría* las cabeceras de las funciones/procedimientos. Esas variables no son parte del programa, sino parte del proceso desarrollo. Si siempre fuéramos a generar el log, por ser requisito, podríamos entonces ponerlo como un parámetro en las operaciones.

5. Análisis de ficheros log

Al igual que en la práctica anterior, el log generado por la ejecución del programa se puede testear en la siguiente dirección:
`https://mizar.unizar.es/pscd`

6. Entrega de la práctica

Una vez la práctica esté terminada, los dos componentes de la pareja deben entregar, cada uno desde su cuenta, el mismo fichero comprimido `practica_4_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_4.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. Los ficheros `ControlCabinas.hpp` y `ControlCabinas.cpp` con la especificación y la implementación del monitor para el control de la concurrencia.
4. El fichero `Makefile_p4` que compila el fuente, generando el ejecutable `practica_4`
5. Todos los demás ficheros requeridos para que la ejecución de `make -f Makefile_p4` 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_4_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_4_NIP1_NIP2.zip`
3. Alternativamente lo podemos hacer desde la terminal como sigue. Una vez creado el directorio `practica_4_NIP1_NIP2` con los ficheros pedidos ejecutamos lo siguiente desde la terminal:

```
zip -r practica_4_NIP1_NIP2.zip practica_4_NIP1_NIP2
```

Con el fin de comprobar que el `zip` contiene todos los ficheros que debe, y organizados adecuadamente, podéis ejecutar el script `pract_4_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_4_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 17 de noviembre deben entregar no más tarde del 27 de noviembre a las 20:00
- Las sesiones del 18 de noviembre deben entregar no más tarde del 28 de noviembre, a las 20:00
- Las sesiones del 24 de noviembre deben entregar no más tarde del 4 de diciembre, a las 20:00
- Las sesiones del 25 de noviembre deben entregar no más tarde del 5 de diciembre, 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  Monitor ControlCabinas
8      ...
9
10     // Pre: 1 ≤ i ≤ N_USER ∧ 1 ≤ j ≤ N_TIMES_USER
11     // Post: "cab" tomará el valor de la cabina ocupada por el usuario "i"
12     // en su "j"-ésima ejecución. Y la cabina queda ocupada
13     operation entraUsuario(integer i,j, REF integer cab)
14         ...
15
16             ADD_EVENT("USER_IN_" + to_string(i) + "," + to_string(j) + "," +
                   ↪ to_string(cab));
17     end
18
19     // Pre: 1 ≤ i ≤ N_USER ∧ 1 ≤ j ≤ N_TIMES_USER ∧ 0 ≤ cab < N_CAB
20
21     // Post: la cabina "cab" ha sido liberada
22     operation saleUsuario(integer i,j,cab)
23         ...
24
25             ADD_EVENT("USER_OUT_" + to_string(i) + "," + to_string(j) + "," +
                   ↪ to_string(cab));
26     end
27
28     operation entraLimpieza(integer j)
29         ...
30
31             ADD_EVENT("CLEANING_IN" + "," + to_string(j));
32     end
33
34     operation saleLimpieza(integer j)
35         ...
36
37             ADD_EVENT("CLEANING_OUT" + "," + to_string(j));
38     end
39 end
40
41 Process usuario(i: 1..N_USER ...)::
42     integer cab //para la cabina que uso
43     ADD_EVENT("USER_BEGIN_" + to_string(i));
44     for j:=1..N_TIMES_USER
```

```
45              ADD_EVENT("USER_WAIT_" + to_string(i) + "," + to_string(j));
46          ControlCabinas.entraUsuario(i,j,cab)
47          //usa la cabina durante un tiempo aleatorio [10,40]
48          ControlCabinas.saleUsuario(i,j,cab)
49        //tiempo aleatorio de espera, [20,50], antes de volver a usa el locutorio
50     end
51     ADD_EVENT("USER_END_" + to_string(i));
52  end
53
54  Process cleaning(...)::
55      ADD_EVENT("CLEANING_BEGIN");
56      for j:=1..N_TIMES_CLEANING
57          //espera PER_CLEANING
58              ADD_EVENT("CLEANING_WAIT" + "," + to_string(j));
59          ControlCabinas.entraLimpieza(j)
60          //tiempo aleatorio, [80,120], simulando ejecución de la limpieza
61          ControlCabinas.saleLimpieza(j)
62     end
63     ADD_EVENT("CLEANING_END");
64  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.

```
1  ID,ts,threadID,ticket,event,a1,a2,a3,...
2  id_139874549184320,1634567931851068300,139874549184320,1,BEGIN_MAIN
3  id_139874549184320,1634567931851279601,139874549180160,2,CLEANING_BEGIN
4  id_139874549184320,1634567931851517419,139874540787456,3,USER_BEGIN_0
5  id_139874549184320,1634567931851658495,139874549180160,4,CLEANING_WAIT,0
6  id_139874549184320,1634567931851688662,139874532394752,5,USER_BEGIN_1
7  ...
8  id_139874549184320,1634567931968453972,139873654662912,59,USER_WAIT_16,1
9  id_139874549184320,1634567931975443603,139873629484800,60,USER_OUT_19,0,3
10 id_139874549184320,1634567931975468747,139873629484800,61,USER_WAIT_19,1
11 id_139874549184320,1634567931975508956,139874132817664,62,USER_IN_8,0,3
12 id_139874549184320,1634567931988559152,139874515609344,63,USER_OUT_3,0,1
13 id_139874549184320,1634567931988574780,139873637877504,64,USER_IN_18,0,1
14 ...
15 id_139874549184320,1634567936150434182,139874107639552,1857,USER_OUT_11,29,0
16 id_139874549184320,1634567936150456993,139874107639552,1858,USER_END_11
17 id_139874549184320,1634567936150655349,139874549184320,1859,END_MAIN
```