Programación Concurrente

Javier Rodríguez Martín

27 de enero de 2016

Índice

1. Introducción

1.1. Objetivo

En este trabajo vamos a implementar un algoritmo que trabaja de manera concurrente. El problema a tratar es uno de los propuestos para programar sobre concurrencia en Java (concretamente el boletín 3, adjunto en el anexo), pero la implementación que vamos a realizar será en C++, de manera que tendremos que trabajar de otra forma a la propuesta.

En líneas generales, se trata de implementar una clase puente y una clase vehículo de tal forma que cada vehículo tenga un hilo propio de ejecución, los cuales comparten el recurso puente de forma concurrente, simulando así el tráfico en dicho puente.

1.2. Dificultades

Los retos que presentan este trabajo de forma personal son los siguientes.

Uso de LaTeX No tengo ninguna experiencia anterior usando este sistema de composición de textos.

Uso de C++ Nunca he programado en C++. Por otro lado, si bien tengo conceptos básicos de C, mis conocimientos de programación se basan más en Java que en C.

Uso de la API POSIX La necesitaremos para implementar los hilos, concretamente necesitaremos profundizar en la biblioteca pthread y hacer uso de sus métodos de creación y sincronización de hilos.

2. Biblioteca *pthread*

Para que nuestro programa se ejecute de forma concurrente, es necesario separar la ejecución del proceso en varios hilos. Para ello necesitaremos usar la API proporcionada por el sistema operativo, de las cuales las más comunes son WIN32/64 y POSIX. Aquí debemos hacer una primera elección, y ya que estamos trabajando desde un sistema basado en Linux vamos a usar la API POSIX. Para ello usaremos la biblioteca *GNU C*, concretamente el archivo *pthread.h*, cuyo nombre viene de POSIX thread, y que nos servirá para trabajar con los hilos.
Un concepto importante a la hora de usar estos hilos es que, si no implementamos un método de sincronización entre ellos, aunque los creemos, estos hilos pueden comportarse de manera indeseada. Como hemos visto en el temario, cuando varios hilos quieren actuar de manera concurrente, si no se sincronizan, pueden dar lugar a condiciones de carrera en los recursos compartidos. El programador es el responsable de que los hilos que él crea se

sincronicen adecuadamente, y pthreads proporciona tres métodos de sincronización para ello, cada uno con sus rutinas:

Joining La subrutina *pthread_join* bloquea al hilo que la invoca hasta que el hilo que se le pasa como argumento termina su ejecución. Esta es la forma más básica de sincronización, pero su sencillez también tiene algo perjudicial: un abuso de este método implica que los hilos no puedan trabajar a la vez, si no como una sucesión. Realmente podrían realizar parte del código de forma simultanea (o turnándose si no hay multiprocesamiento) y al llegar a una sección crítica, realizar el join para esperar que uno termine para que pueda continuar el otro. No es el método que se va a implementar para este trabajo.

Variables *Mutex* Abreviación de Mutual Exclusion, estas variables proporcionan una forma de protección para los datos compartidos por los hilos. Actúan como cerrojos, de forma que cuando un hilo actúa sobre una sección crítica, cierra el cerrojo, de forma que los otros hilos que intenten acceder a esta sección tendrán que esperar a que el proceso abra de nuevo dicho cerrojo. Es una forma de asegurar que parte del código se ejecuta de forma atómica y es un elemento que necesitaremos conjunto al último método de sincronización.

Variables de Condición Estas variables proporcionan una forma de sincronización en base al valor de algún dato. Siempre se usan conjuntamente con variables mutex y es lo que necesitaremos para controlar el acceso al puente según el sentido de los vehículos. Si no hiciéramos uso de este método, tendríamos que realizar una espera ocupada de los hilos preguntando por el estado del puente. Gracias a una variable de condición podremos bloquear los hilos y hacer que esperen a que el puente tenga el sentido que ellos necesitan, y entonces les avise.

A continuación describiremos las diferentes rutinas que vamos a necesitar de *pthread*.

2.1. Creación y finalización de hilos

pthread_create(hilo,attr,rutina,arg) Crea un hilo. Tiene cuatro parámetros de entrada. El primero es un identificador único para el nuevo hilo, dado por la subrutina. El segundo sirve para definir los atributos del hilo, se puede dejar a NULL para usar los atributos por defecto. El tercero es la rutina que el hilo va a ejecutar una vez que lo creemos, y el cuarto sirve para pasar un parámetro a esa rutina. Este argumento debe cambiarse al tipo void* (mediante *casting*), pues es el tipo de argumento que se espera. Si quisiéramos pasar más de un argumento tendría que ser a través de un objeto con varias propiedades, o un struct.

pthread_exit(estado) Sirve para finalizar la ejecución de un hilo. Se puede definir como parámetro el estado de terminación del hilo, que sirve para trabajar con joins. Como no vamos a hacerlo, lo dejaremos a null. Si un hilo finaliza con esta rutina, no destruye los hilos que dependan de él, como lo haría al finalizar normalmente, si no que en realidad se mantiene bloqueado hasta que el resto de hilos que dependen de él hayan finalizado. Habitualmente se usa en el main para que al finalizar su código el programa no termine, si no que deje actuar a los hilos creados en este.

2.2. Mutex y variables de condición

pthread_mutex_lock(mutex) Tiene como parámetro una variable del tipo *pthread_mutex_t*, y la rutina cerrará este cerrojo. Si ya estaba cerrado por otro hilo, la llamada bloqueará al hilo hasta que se abra la variable mutex.

pthread_mutex_unlock(mutex) Abre la variable mutex que se le pasa como parámetro. Devolverá un error si ya estaba abierta o la variable mutex estaba cerrada por otro hilo. Al abrir la variable mutex, como no vamos a implementar un planificador de prioridades de hilos, el sistema operativo decidirá que hilo es el que se desbloquea de todos los que están esperando a que se abra el cerrojo, de manera más o menos aleatoria.

pthread_cond_wait(condición, mutex) Se le pasa como parámetros una variable del tipo *pthread_cond_t* y una variable mutex. Bloquea el hilo que llama a la rutina hasta que recibe una señal de la variable de condición. Se debe usar mientras la variable mutex esta cerrada, y automáticamente abrirá esta variable mutex. Cuando reciba la señal, la variable mutex se bloqueará automáticamente para este hilo.

pthread_cond_signal(condición) Envía una señal a algún hilo que este esperando por la condición que se pasa como parámetro. Se debe usar después de haber cerrado la variable mutex, y, una vez que se ha llamado a la rutina, se debería abrir esta misma variable mutex para que el hilo que recibe la señal pueda continuar su ejecución, ya que como hemos dicho antes automáticamente cerrará la variable mutex, y como la señal se ha enviado con la mutex ya cerrada, se bloqueará esperando a que vuelva a abrirse.

pthread_cond_broadcast(condición) Se usa exactamente igual que la anterior, pero envía la señal a todos los hilos que estén esperando por la condición. Normalmente, esta es la rutina necesaria cuando más de un hilo espera a la misma señal, y es la que usaremos nosotros.

Es muy importante usar correctamente las variables de exclusión mutua al usar variables de condición, pues un mal uso podría provocar que el hilo no se bloquee al usar la rutina pthread_cond_wait o que no se desbloquee después de usar la rutina pthread_cond_signal (o pthread_cond_broadcast).

Tanto las variables del tipo pthread_mutex_t como las del tipo pthread_cond_t necesitan ser inicializadas, y la biblioteca pthread proporciona dos métodos para hacerlo. Nosotros usaremos sus variables estáticas PTHREAD_MUTEX_INITIALIZER y PTHREAD_COND_INITIALIZER respectivamente para ello. También deberían ser destruidas cuando ya no sean necesarios para liberar los recursos con las rutinas pthread_mutex_destroy y pthread_cond_destroy. En principio, nosotros vamos a usarlas durante toda la ejecución de nuestro código, así que vamos obviar este paso. Si quisiéramos destruirlos, deberíamos hacer que el main espere con pthread_join al resto de los hilos, y después llamar a dichas funciones, pues con pthread_exit, que es el que estamos usando, espera a que los demás finalicen también, pero luego no podemos realizar ninguna acción en el main.

3. Implementación

En este apartado vamos a comentar las partes más interesantes de la implementación, mostrando partes del código adjunto. Nos centraremos en los hilos y los métodos de sincronización comentados, y en cambio no vamos a profundizar en cosas como la creación de objetos en C++ o los punteros a variables. Si bien ha costado tiempo y esfuerzo comprender bien los detalles de estas partes, en realidad no son el objetivo de nuestro trabajo, si no meras herramientas que necesitábamos usar, y en realidad bastante básicas. El código completo se adjunta en el anexo.

3.1. Sin hilos de ejecución

Antes de llegar a la implementación final hemos tenido que hacer varios arreglos para ir puliendo algunos detalles. Primero implementamos las clases de los objetos y del main sin hacer uso de hilos de ejecución. Veamos ambas clases para tener una idea de como quedaría esta primera versión de los objetos.

Puente.h:

```
1 #ifndef PUENTE_H
2 #define PUENTE_H
3 #include <string>
4
5 using namespace std;
6
7 class Puente
8 {
```

```
9  private:
10     int nVehiculos;
11     int extremoAbierto;
12 public:
13     Puente();
14     void entrar (string matricula, int extremo);
15     void salir (string matricula);
16     static const int A = 0;
17     static const int B = 1;
18 };
19
20 #endif // PUENTE_H
```

Puente.cpp:

```
1  #include <puente.h>
2  #include <iostream>
3
4  Puente::Puente()
5  {
6      nVehiculos = 0;
7      extremoAbierto = -1; // -1 indica que ambos extremos estan ↩
           abiertos, 0 indica que el extremo A esta abierto y 1 ↩
           que el extremo B esta abierto.
8  }
9
10 void Puente::entrar (string matricula, int extremo)
11 {
12     string strExtremo = extremo?"B":"A";
13     string strExtremoOpuesto = extremo?"A":"B";
14
15     while (extremoAbierto != extremo && extremoAbierto != -1);
16
17     if (extremoAbierto == -1)
18     {
19         cout << "** Extremo "<< strExtremoOpuesto << " cerrado.↩
                Sentido "<< strExtremo << " -> " << ↩
               strExtremoOpuesto << " **" << endl;
20         extremoAbierto = extremo;
21     }
22
23     nVehiculos++;
24
25     cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
           \" entra por el extremo " << strExtremo <<
26             ". Hay " << nVehiculos << " coche";
27     if(nVehiculos != 1)
28         cout << "s";
29     cout << " en el puente." << endl;
30
31
32 }
33
```

```
34 void Puente::salir (string matricula)
35 {
36     string strExtremoOpuesto = extremoAbierto?"A":"B";
37     nVehiculos --;
38 
39     cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
           \" sale por el extremo " << strExtremoOpuesto <<
40             ". Hay " << nVehiculos << " coche";
41     if(nVehiculos != 1)
42         cout << "s";
43     cout << " en el puente." << endl;
44 
45     if (nVehiculos == 0)
46     {
47         extremoAbierto = -1;
48         cout << "** Puente ívaco, extremos abiertos. **" << ↩
               endl;
49     }
50 }
```

Vehiculo.h:

```
1 #ifndef VEHICULO_H
2 #define VEHICULO_H
3 #include <puente.h>
4 
5 class Puente;
6 
7 class Vehiculo
8 {
9 private:
10     Puente *puente;
11     string matricula;
12     struct timespec tim, tim2;
13 
14 public:
15     Vehiculo(string matricula, Puente *puente);
16     void cruzaPuente();
17 };
18 
19 #endif // VEHICULO_H
```

Vehiculo.cpp:

```
1 #include <vehiculo.h>
2 #include <stdlib.h>
3 
4 Vehiculo::Vehiculo(string matricula, Puente *puente)
5 {
6     this->puente = puente;
7     this->matricula = matricula;
8     cruzaPuente();
```

```
9  }
10
11
12 void Vehiculo::cruzaPuente()
13 {
14     int extremo = (rand() %2)?Puente::A:Puente::B;
15
16     puente->entrar(matricula, extremo);
17
18     tim.tv_sec  = 0;
19     tim.tv_nsec = rand()%500000000;  // 0.5 segundos en ↩
           nanosegundos
20     nanosleep(&tim , &tim2);
21
22     puente->salir(matricula);
23 }
```

main.cpp:

```
1  #include <vehiculo.h>
2  int main()
3  {
4
5      Puente puente;
6      Vehiculo vehiculo ("0000 AAA", &puente);
7      return 0;
8  }
```

Lógicamente en este punto solo puede haber como máximo un vehículo en el puente, y además ninguno puede estar esperando, por lo que podríamos ahorrarnos unas cuantas cosas, como el atributo nVehiculos o la lógica de esperar si el sentido no es el del vehículo, implementado en el método entrar() como una espera ocupada en este momento (pero sin ninguna utilidad, ya que al no poder haber más de un coche en el puente, el que pase siempre dejará al salir los dos extremos libres).

3.2. Creación de hilos

Una vez que estas clases funcionaban correctamente (al crear un puente y después un vehículo que pasara por ese puente, efectivamente cruzaba el puente entrando por un extremo y saliendo por el otro), pasamos a la implementación de la concurrencia. Para ello primero modificamos la clase vehículo y le añadimos un método estático que sería el que le pasaríamos a la rutina pthread_create como parámetro en el main. Además también añadimos un método estático para obtener matriculas aleatorias, facilitando así la creación de vehículos.

Como podemos observar en la segunda linea, en el puntero a void pasamos en realidad un objeto de tipo puntero a puente, por lo que tenemos que

cambiarle el tipo antes de usarlo. Una vez hecho esto, se pasa como parámetro con la matrícula escogida aleatoriamente en el método getRandMatricula().

De este último método quizá es necesario explicar como funcionan las lineas 10 y 15. Por un lado rand() genera un número aleatorio comprendido entre 0 y 2147483647, y por otro, dado un entero al tipo char es lo mismo que decirle el número ASCII del carácter que queremos. Por tanto, para generar caracteres de números entre 0 y 9, necesitamos un número aleatorio entre el 48 y el 57, ambos incluidos, que son los que corresponden en ASCII con ellos. Al dividir el número aleatorio entre el rango, nos quedará un número entre el 0 y el 9, y para que se corresponda con el que necesitamos le sumamos 48. De la misma forma hemos generado un carácter aleatorio en mayúsculas.

Los nuevos métodos incluidos en vehiculo.cpp (cuyas declaraciones han sido incluidas en el fichero vehiculo.h) son:

```
1  void *Vehiculo::creaVehiculo(void* object){
2      Puente* puente = reinterpret_cast<Puente*>(object);
3      Vehiculo coche (Vehiculo::getRandMatricula(), puente);
4  }
5
6  string Vehiculo::getRandMatricula(){
7
8      string str = "";
9      for (int i=0; i<4; i++){
10         char numMatricula = rand()%(58-48)+48;
11         str += numMatricula;
12     }
13     str += " ";
14     for (int i=0; i<3; i++){
15         char letraMatricula = rand()%(91-65)+65;
16         str += letraMatricula;
17     }
18     return str;
19 }
```

En la clase main simplemente añadimos la llamada a la rutina de creación de hilos, sustituyendo la llamada al constructor de vehículos. Además lo hemos incluido en un bucle para que se creen varios vehículos:

```
1  int main()
2  {
3      Puente puente();
4      for (int i=0; i<100; i++){
5          pthread_t tid;
6          pthread_create(&tid, NULL,Vehiculo::creaVehiculo, &↩
               puente);
7      }
8      pthread_exit(NULL);
9      return 0;
10 }
```

Podemos observar que además hemos añadido una llamada a la rutina pthread_exit, pues si no lo hiciéramos, al llegar a este punto, el programa finalizaría independientemente de que los hilos hayan finalizado.

Si ejecutáramos el programa tal cual podríamos ver que es lo que pasa si no realizamos una implementación de la sincronización. No solo la ejecución de un hilo provoca condiciones de carrera en los atributos del puente, provocando que un vehículo aparentemente pueda salir por el mismo extremo por el que entró, si no que también, al escribir por pantalla, un hilo puede escribir en mitad de una frase de otro hilo, pues después de todo, la pantalla es otro recurso compartido en el que también se producirían errores por la falta de sincronización.

3.3. Exclusión mutua

Para implementar la sincronización vamos a necesitar una variable mutex. Hemos elegido implementarla en el tipo puente, ya que es el recurso que va a ser compartido, pero podríamos haberlas implementados dentro de una nueva clase estática, por ejemplo, de la que hagan uso los vehículos. En la cabecera de la clase puente definimos las variables necesarias y las inicializamos como explicamos en el apartado correspondiente.

En la parte privada de la cabecera añadimos:

```
1     pthread_mutex_t sentido_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Y en la pública añadimos lo siguiente para que los vehículos puedan usarlas:

```
1     pthread_mutex_t* getMutex();
```

Cuya implementación es simplemente:

```
1 pthread_mutex_t* Puente::getMutex()
2 {
3     return &sentido_mutex;
4 }
```

Cada vez que un vehículo haga una llamada a un método de la clase puente debe hacerlo sin permitir que los demás puedan hacerlo. Para ello hacemos uso de las rutinas pthread_mutex_lock y pthread_mutex_unlock antes y después de llamar a los métodos entrar() y salir() dentro de cruzaPuente(). Este último método quedaría así:

```
1 void Vehiculo::cruzaPuente()
2 {
```

```
3     int extremo = (rand() % 2)?Puente::A:Puente::B;
4
5     pthread_mutex_lock(puente->getMutex());
6     puente->entrar(matricula, extremo);
7     pthread_mutex_unlock(puente->getMutex());
8
9     tim.tv_sec  = 0;
10    tim.tv_nsec = rand()%500000000;  // 0.5 segundos en ↩
          nanosegundos
11    nanosleep(&tim , &tim2);
12
13    pthread_mutex_lock(puente->getMutex());
14    puente->salir(matricula);
15    pthread_mutex_unlock(puente->getMutex());
16
17    pthread_exit(NULL);
18 }
```

Con esto la clase vehículo queda terminada, tal y como la presentamos en la versión final en el anexo, pero como podemos observar, el problema no esta solucionado del todo. De hecho, en cuanto un vehículo al entrar tenga que esperar, el programa se bloqueará en una espera ocupada, pues el hilo esperará indefinidamente a que el puente cambie su estado sin que esto pueda pasar, pues él mismo tiene bloqueado el recurso puente y ningún otro hilo puede modificarlo.

3.4. Variables de condición

La solución es obvia, si el hilo no puede usar el recurso por el estado de este, en vez de seguir preguntando por él, que se bloquee, libere el recurso y espere a que este cambie de estado. Cuando pueda usarlo, inmediatamente bloqueará de nuevo el recurso y volverá a preguntar sobre su estado. De esta forma es como usaremos las variables de condición, pues esto se hace automáticamente con la rutina pthread_cond_wait. Primero tenemos que declarar e inicializar la variable de condición. Al igual que hicimos con la variable mutex, lo haremos en la cabecera de la clase puente. Como la clase vehículo no necesita usarla, no necesitaremos método publico que la devuelva, así que la añadimos en la parte privada:

```
1     pthread_cond_t sentido_threshold_cv = ↩
          PTHREAD_COND_INITIALIZER;
```

Una vez hecho esto, en el método entrar() vamos a perfilar un poco la espera:

```
1 void Puente::entrar (string matricula, int extremo)
2 {
3     string strExtremo = extremo?"B":"A";
```

```
4     string strExtremoOpuesto = extremo?"A":"B";
5
6     while (extremoAbierto != extremo && extremoAbierto != -1)
7     {
8         cout << "El ívehculo con la ímatrcula \"" << matricula ↩
              << "\" espera en el extremo " << strExtremo << endl↩
              ;
9         pthread_cond_wait(extremo?&sentido_B_threshold_cv:&↩
              sentido_A_threshold_cv, &sentido_mutex);
10
11    }
12
13    if (extremoAbierto == -1)
14    {
15        cout << "** Extremo "<< strExtremoOpuesto << " cerrado.↩
               Sentido "<< strExtremo << " -> " << ↩
              strExtremoOpuesto << " **" << endl;
16        extremoAbierto = extremo;
17    }
18
19    nVehiculos++;
20
21    cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
          \" entra por el extremo " << strExtremo <<
22              ". Hay " << nVehiculos << " coche";
23    if(nVehiculos != 1)
24         cout << "s";
25    cout << " en el puente." << endl;
26
27
28 }
```

Para que esto funcione, tenemos que implementar también uno de los métodos dados para desbloquear un hilo bloqueado por una variable de condición. En nuestro caso, como prevemos que muchos hilos se bloquearán en este punto, usaremos pthread_cond_broadcast, pues si no, tan solo uno de los hilos bloqueados pasaría a estar desbloqueado, y sería por tanto el único que pasaría por el puente antes de volver a desbloquear a otro: en definitiva, pasarían de uno en uno por el puente. La modificación que debemos hacer es muy simple, en el método salir():

```
1
2  void Puente::salir (string matricula)
3  {
4      string strExtremo = extremoAbierto?"A":"B";
5      nVehiculos --;
6      cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
           \" sale por el extremo " << strExtremo <<
7               ". Hay " << nVehiculos << " coche";
8      if(nVehiculos != 1)
9          cout << "s";
```

```
10      cout << " en el puente." << endl;
11
12      if (nVehiculos == 0)
13      {
14        pthread_cond_broadcast(&sentido_threshold_cv);
15          extremoAbierto = -1;
16          cout << "** Puente vacio, extremos abiertos. **" << ↩
                endl;
17      }
18  }
```

Tan solo hemos añadido la linea 15 a lo que ya teníamos. En este punto podríamos dar por finalizada la implementación, pero se propone una mejora de la clase puente que vamos a realizar y que da pie a una segunda mejora.

3.5. Límite de vehículos

Ahora mismo el puente, tal y como funciona, deja pasar primero a todos los vehículos que vayan en una dirección y luego a los que esperan en el otro extremo. Si no pararan de llegar vehículos, los que están en espera quedarían así de forma indefinida. Para evitarlo vamos a establecer un limite de coches que pueden pasar de manera consecutiva dentro del puente, de forma que si se llega a dicho limite y hay coches esperando en el otro extremo, se espere a que el puente se vacíe, y entonces se dará paso a los que estaban esperando en el otro extremo. Solo tendremos que modificar la clase puente, añadiendo algunos atributos, y en la clase main añadir el máximo de vehículos consecutivos como parámetro al crear el puente. Los atributos que vamos a añadir son un máximo de vehículos, el número de vehículos consecutivos que han pasado, y el número de vehículos que esperan en A y en B. Los declaramos en puente.h y los inicializamos en el constructor:

```
1  Puente::Puente(int max)
2  {
3      maxVehiculos = max;
4      nVehiculos = 0;
5      extremoAbierto = -1; // -1 indica que ambos extremos estan ↩
           abiertos, 0 indica que el extremo A esta abierto y 1 ↩
           que el extremo B esta abierto.
6      consecutivos = 0;
7      esperandoEnA = 0;
8      esperandoEnB = 0;
9  }
```

En el método entrar() tendremos que añadir el vehículo a la espera del extremo correspondiente, hacer que espere aun si el sentido del puente es el mismo que el suyo en caso de que se haya alcanzado o superado el máximo de vehículos consecutivos y haya algún vehículo esperando en el extremo contrario, y por último añadir el coche a los que han pasado de forma consecutiva

y eliminarlo de la espera en su extremo en caso de que entre:

```
1  void Puente::entrar (string matricula, int extremo)
2  {
3
4      int esperandoEnExtremoOpuesto = extremo?esperandoEnA:↩
           esperandoEnB;
5      string strExtremo = extremo?"B":"A";
6      string strExtremoOpuesto = extremo?"A":"B";
7      extremo? esperandoEnB++ : esperandoEnA++;
8
9      while ((extremoAbierto != extremo && extremoAbierto != -1) ↩
           || (consecutivos >= maxVehiculos && ↩
           esperandoEnExtremoOpuesto))
10     {
11         cout << "El ívehculo con la ímatrcula \"" << matricula ↩
               << "\" espera en el extremo " << strExtremo << endl↩
               ;
12         pthread_cond_wait(extremo?&sentido_B_threshold_cv:&↩
               sentido_A_threshold_cv, &sentido_mutex);
13
14     }
15
16     if (extremoAbierto == -1)
17     {
18         cout << "** Extremo "<< strExtremoOpuesto << " cerrado.↩
                Sentido "<< strExtremo << " -> " << ↩
               strExtremoOpuesto << " **" << endl;
19         extremoAbierto = extremo;
20     }
21
22     consecutivos++;
23     nVehiculos++;
24     extremo? esperandoEnB-- : esperandoEnA--;
25
26     cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
           \" entra por el extremo " << strExtremo <<
27             ". Hay " << nVehiculos << " coche";
28     if(nVehiculos != 1)
29         cout << "s";
30     cout << " en el puente." << endl;
31
32
33 }
```

Lo último que necesitamos es añadir en el método salir() una puesta a 0 del contador de vehículos consecutivos en caso de que no queden vehículos en el puente:

```
1    void Puente::salir (string matricula)
2  {
```

```
3     string strExtremoOpuesto = extremoAbierto?"A":"B";
4     nVehiculos --;
5     cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
          \" sale por el extremo " << strExtremoOpuesto <<
6               ". Hay " << nVehiculos << " coche";
7     if(nVehiculos != 1)
8         cout << "s";
9     cout << " en el puente." << endl;
10
11
12    if (nVehiculos == 0)
13    {
14      pthread_cond_broadcast(&sentido_threshold_cv);
15        extremoAbierto = -1;
16        consecutivos = 0;
17        cout << "** Puente ívaco, extremos abiertos. **" << ↩
              endl;
18    }
19 }
```

Si bien con estos cambios ya cumple bien su función y permite hacer cambios de sentido cada cierto número de coches que lleguen, una ejecución de este programa y un poco de atención a la traza que dibuja es suficiente para que nos preguntemos algo. ¿Es realmente necesario que la señal de la variable de condición le llegue a todos los vehículos que están esperando?

3.6. Segunda variable de condición

La respuesta es no, lógicamente. En un primer intento por mejorar esto, vamos a crear una segunda variable de condición, de forma que una variable sirva para un extremo y la otra para el otro extremo. De esta forma, los vehículos que esperan en el extremo A esperaran a una señal diferente que los que esperan en el otro extremo. Cuando el puente cambie de sentido enviará la señal solo al extremo que corresponda, dejando a los del otro extremo aun en espera. Esto es muy fácil de implementar, solo tenemos que crear una nueva variable de condición, declarada e inicializada en la cabecera, y modificar la espera en el método entrar() y la señal en el método salir(). Veamos las partes modificadas de estos dos métodos. En puente.h:

```
1     pthread_cond_t sentido_A_threshold_cv = ↩
          PTHREAD_COND_INITIALIZER;
2     pthread_cond_t sentido_B_threshold_cv = ↩
          PTHREAD_COND_INITIALIZER;
```

En entrar(), dentro del while:

```
1         while ((extremoAbierto != extremo && extremoAbierto != ↩
              -1) || (consecutivos >= maxVehiculos && ↩
```

```
            esperandoEnExtremoOpuesto))
    {
        cout << "El ívehculo con la ímatrcula \"" << matricula ↩
            << "\" espera en el extremo " << strExtremo << endl↩
            ;
        pthread_cond_wait(extremo?&sentido_B_threshold_cv:&↩
            sentido_A_threshold_cv, &sentido_mutex);

    }
```

En salir(), dentro del if que pregunta si han salido todos los vehiculos del puente:

```
       if (nVehiculos == 0)
    {
        if (extremoAbierto == A && esperandoEnB)
         pthread_cond_broadcast(&sentido_B_threshold_cv);

        else if (esperandoEnA)
            pthread_cond_broadcast(&sentido_A_threshold_cv);

        extremoAbierto = -1;
        consecutivos = 0;
        cout << "** Puente ívaco, extremos abiertos. **" << ↩
            endl;
    }
```

Si ejecutamos de nuevo el programa podemos ver en la traza como se han reducido notablemente los vehículos que aparecen esperando de nuevo cada vez que el puente alcanza el máximo número de vehículos consecutivos. Aun así podemos pulirlo un poco más.

3.7. Uso de signal repetidas veces en vez de broadcast

Si tenemos en cuenta que hemos contado el número de vehículos que esperan en cada extremo, y que hay un número máximo de vehículos consecutivos que pueden pasar, podemos avisar solo al número de vehículos que van a poder pasar, evitando que ningún vehículo tenga que quedarse a la espera de nuevo por haber desbloqueado más hilos de los que realmente podían usar el recurso. La modificación va solamente en el método salir(), dentro del mismo if que antes, y con esta queda el código tal y como se muestra en el anexo:

```
    if (nVehiculos == 0)
    {
        if (extremoAbierto == A && esperandoEnB)
        {
            maxBroadcast = esperandoEnB<maxVehiculos? ↩
                esperandoEnB : maxVehiculos;
```

```
6               for (int i = 0; i < maxBroadcast; i++)
7                   pthread_cond_signal(&sentido_B_threshold_cv);
8           }
9
10          else if (esperandoEnA)
11          {
12              maxBroadcast = esperandoEnA<maxVehiculos? ↩
                    esperandoEnA : maxVehiculos;
13              for (int i = 0; i < maxBroadcast; i++)
14                  pthread_cond_signal(&sentido_A_threshold_cv);
15          }
16
17          extremoAbierto = -1;
18          consecutivos = 0;
19          cout << "** Puente ívaco, extremos abiertos. **" << ↩
                endl;
20      }
```

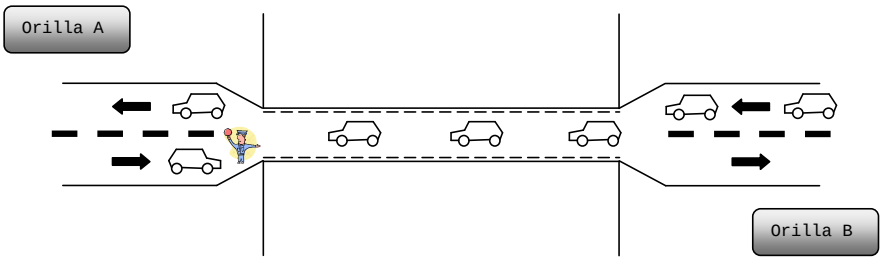
4. Anexo

4.1. Enunciado

Boletín 3 — Concurrencia con Java

Ejercicio 1

Un puente une dos orillas, a las que llamaremos A y B. Dicho puente es de un único carril, por lo que sólo pueden circular coches en un único sentido. Por ello, existe un mecanismo de control del tráfico de forma que cuando no hay ningún vehículo en el puente, el primer vehículo que llegue puede cruzarlo. Mientras este vehículo está cruzando, otros vehículos que lleguen en el mismo sentido pueden entrar también en el puente, pero no los que lleguen en sentido contrario, que tendrán que esperar a que el puente quede libre para poder pasar.



Impleméntese la lógica de dicho puente mediante una clase Java `Puente` que disponga de los siguientes métodos:

- `void entrar (Vehiculo v, int extremo)`: simula la entrada de un vehículo `v` por uno de los extremos (constantes `Puente.A` o `Puente.B`) del puente. Si por el otro extremo ha entrado algún vehículo que aún no ha salido, esperará hasta que hayan salido todos los que circulen en sentido contrario.
- `void salir (Vehiculo v)`: simula la salida de un vehículo `v`. Se sobreentiende que el extremo por el que sale es el contrario al por el cual entró.

Una vez creada la clase `Puente`, escriba una clase `Vehículo` cuyos objetos contengan un hilo propio de ejecución, que simule el tráfico a través del puente, de forma que cada vehículo elija un sentido aleatorio de circulación, entrando por el extremo correspondiente del puente y saliendo por el extremo contrario tras un tiempo aleatorio entre 0 y 0,5 segundos. Además de ello, y para facilitar la traza del programa, cada vehículo tendrá una matrícula que se establecerá en el constructor y que se podrá consultar mediante el método:

- `String matricula()`: proporciona la matrícula del vehículo.

Escriba un programa que cree el puente y un número determinado de objetos de la clase vehículos que crucen el puente en sentidos aleatorios, y compruebe el funcionamiento del programa.

Observaciones:

- Es buena idea que la clase vehículo acepten en su constructor una referencia al objeto puente para que “sepan” qué puente han de cruzar.
- También puede ser buena idea separar la interfaz de los vehículos en una interfaz IVehículo implementada por todo vehículo, y la interfaz del puente en una interfaz IPuente implementada por todo puente, de forma que por ejemplo las operaciones de entrar y salir

del puente acepten referencias a esta interfaz en lugar de a la propia clase vehículo. Verá la aplicación de esto al construir la versión distribuida de este ejercicio.

- Para poder seguir la traza del programa podría ser buena idea que el puente muestre por la salida estándar las matrículas de los vehículos que entran y salen en cada sentido, así como de los que están esperando para entrar si procede. Tenga en cuenta las consideraciones sobre el entrelazado de información en la salida vista anteriormente.
- Es cierto que si un vehículo V1 entra por el puente antes que un vehículo V2 (por supuesto en el mismo sentido), y V2 sale antes que V1, es porque V2 debería haber saltado sobre V1. Debe tratarse del coche fantástico, no se preocupe por ello ☺.
- Para generar una espera aleatoria de entre 0 y N-1 milisegundos, puede usar el siguiente código:

```
long delay=(long) (Math.random()*N);
Thread.currentThread().sleep(delay); //o bien Thread.sleep(delay)
```

Ejercicio 2

Habrá comprobado que en muchas ejecuciones del programa, el primer vehículo que entra en el puente establece el sentido de circulación y a partir de ese momento, todos los vehículos que llegan en sentido contrario tienen que esperar hasta que pasa el último vehículo en el sentido inicial, pasando dichos vehículos en sentido contrario cuando el puente por fin queda libre. Esto significa que nuestro puente no es *justo*, pues puede provocar *inanición* (también llamado aplazamiento indefinido) de los vehículos que pretenden cruzar en un sentido mientras no dejen de llegar vehículos en sentido contrario.

Escriba una clase `PuenteJusto` que modifique el comportamiento de la clase `Puente` de forma que si tras entrar en el puente un número determinado de vehículos consecutivos en un mismo sentido, existen vehículos esperando a cruzar en sentido contrario, se invierta el sentido de circulación dando preferencia a dichos vehículos.

Observaciones:

- Tenga en cuenta que al invertir el sentido del puente deberá dejar salir a los vehículos que actualmente circulan por él, antes de permitir entrar a vehículos en el sentido contrario.

4.2. Código fuente

main.cpp:

```
#include <vehiculo.h>
int main()
{


    Puente puente(20);
    for (int i=0; i<100; i++){
        pthread_t tid;
        pthread_create(&tid, NULL,Vehiculo::creaVehiculo, &puente);
    }
    pthread_exit(NULL);
    return 0;
}
```

puente.h:

```
#ifndef PUENTE_H
#define PUENTE_H
#include <string>

using namespace std;

class Puente
{
private:
    int maxVehiculos;
    int nVehiculos;
    int extremoAbierto;
    int consecutivos;
    int esperandoEnA;
    int esperandoEnB;
    pthread_mutex_t sentido_mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t sentido_A_threshold_cv = PTHREAD_COND_INITIALIZER;
    pthread_cond_t sentido_B_threshold_cv = PTHREAD_COND_INITIALIZER;
public:
    Puente(int max);
    void entrar (string matricula, int extremo);
    void salir (string matricula);
    static const int A = 0;
    static const int B = 1;
    pthread_mutex_t* getMutex();
};

#endif // PUENTE_H
```

puente.cpp:

```
#include <puente.h>
#include <iostream>

Puente::Puente(int max)
{
    maxVehiculos = max;
    nVehiculos = 0;
    extremoAbierto = -1; // -1 indica que ambos extremos estan abiertos, 0 indica que el extremo A esta abierto y 1 que el extremo B esta abierto.
    consecutivos = 0;
    esperandoEnA = 0;
    esperandoEnB = 0;
}

void Puente::entrar (string matricula, int extremo)
{

    int esperandoEnExtremoOpuesto = extremo?esperandoEnA:esperandoEnB;
    string strExtremo = extremo?"B":"A";
    string strExtremoOpuesto = extremo?"A":"B";
    extremo? esperandoEnB++ : esperandoEnA++;

    while ((extremoAbierto != extremo && extremoAbierto != -1) || (consecutivos >= maxVehiculos && esperandoEnExtremoOpuesto))
    {
        cout << "El ívehculo con la ímatrcula \"" << matricula << "\" espera en el extremo " << strExtremo << endl;
        pthread_cond_wait(extremo?&sentido_B_threshold_cv:&sentido_A_threshold_cv, &sentido_mutex);

    }

    if (extremoAbierto == -1)
    {
        cout << "** Extremo "<< strExtremoOpuesto << " cerrado. Sentido "<< strExtremo << " -> " << strExtremoOpuesto << " **" << endl;
        extremoAbierto = extremo;
    }

    consecutivos++;
    nVehiculos++;
    extremo? esperandoEnB-- : esperandoEnA--;

    cout << "El ívehculo con la ímatrcula \"" << matricula << "\" entra por el extremo " << strExtremo <<
        ". Hay " << nVehiculos << " coche";
```

```
41      if(nVehiculos != 1)
42          cout << "s";
43      cout << " en el puente." << endl;
44
45
46  }
47
48  void Puente::salir (string matricula)
49  {
50      string strExtremoOpuesto = extremoAbierto?"A":"B";
51      nVehiculos --;
52      cout << "El ívehculo con la ímatrcula \"" << matricula << "↩
            \" sale por el extremo " << strExtremoOpuesto <<
53              ". Hay " << nVehiculos << " coche";
54      if(nVehiculos != 1)
55          cout << "s";
56      cout << " en el puente." << endl;
57      int maxBroadcast;
58      if (nVehiculos == 0)
59      {
60          if (extremoAbierto == A && esperandoEnB)
61          {
62              maxBroadcast = esperandoEnB<maxVehiculos? ↩
                    esperandoEnB : maxVehiculos;
63              for (int i = 0; i < maxBroadcast; i++)
64                  pthread_cond_signal(&sentido_B_threshold_cv);
65          }
66
67          else if (esperandoEnA)
68          {
69              maxBroadcast = esperandoEnA<maxVehiculos? ↩
                    esperandoEnA : maxVehiculos;
70              for (int i = 0; i < maxBroadcast; i++)
71                  pthread_cond_signal(&sentido_A_threshold_cv);
72          }
73
74          extremoAbierto = -1;
75          consecutivos = 0;
76          cout << "** Puente ívaco, extremos abiertos. **" << ↩
                endl;
77      }
78  }
79
80  pthread_mutex_t* Puente::getMutex()
81  {
82      return &sentido_mutex;
83  }
```

vehiculo.h:

```
1  #ifndef VEHICULO_H
2  #define VEHICULO_H
3  #include <puente.h>
```

```
4
5  class Puente;
6
7  class Vehiculo
8  {
9  private:
10     Puente *puente;
11     string matricula;
12     struct timespec tim, tim2;
13
14 public:
15     static string getRandMatricula();
16     static void *creaVehiculo(void* object);
17     Vehiculo(string matricula, Puente *puente);
18     void cruzaPuente();
19
20
21 };
22
23 #endif // VEHICULO_H
```

vehiculo.cpp:

```
1  #include <vehiculo.h>
2  #include <stdlib.h>
3
4  Vehiculo::Vehiculo(string matricula, Puente *puente)
5  {
6      this->puente = puente;
7      this->matricula = matricula;
8      cruzaPuente();
9
10 }
11
12
13 void Vehiculo::cruzaPuente()
14 {
15     int extremo = (rand() % 2)?Puente::A:Puente::B;
16
17     pthread_mutex_lock(puente->getMutex());
18     puente->entrar(matricula, extremo);
19     pthread_mutex_unlock(puente->getMutex());
20
21     tim.tv_sec  = 0;
22     tim.tv_nsec = rand()%500000000;  // 0.5 segundos en ↩
           nanosegundos
23     nanosleep(&tim , &tim2);
24
25     pthread_mutex_lock(puente->getMutex());
26     puente->salir(matricula);
27     pthread_mutex_unlock(puente->getMutex());
28
29     pthread_exit(NULL);
```

```
30 }
31
32 void *Vehiculo::creaVehiculo(void* object){
33     Puente* puente = reinterpret_cast<Puente*>(object);
34     Vehiculo vehiculo (Vehiculo::getRandMatricula(), puente);
35 }
36
37 string Vehiculo::getRandMatricula(){
38
39     string str = "";
40     for (int i=0; i<4; i++){
41         char numMatricula = rand()%(58-48)+48;
42         str += numMatricula;
43     }
44     str += " ";
45     for (int i=0; i<3; i++){
46         char letraMatricula = rand()%(91-65)+65;
47         str += letraMatricula;
48     }
49     return str;
50 }
```

Bibliografía

[1] Blaise Barney, *POSIX Threads Programming*, https://computing.llnl.gov/tutorials/pthreads/, 2015.

[2] *C++ Language*, http://www.cplusplus.com/doc/tutorial/.

[3] *Stack Overflow community*, http://stackoverflow.com/.