Entrega voluntaria: Tres ejercicios con monitores.

José Antonio Álvarez

24 de octubre de 2017

En este documento estudiamos tres ejercicios de monitores algo alternativos. El primero estudia un problema que abordaremos con dos monitores distintos y cuyas condiciones son muy dispares.

En el segundo veremos un ejemplo de recurso compartido al que acceden dos tipos de procesos, de forma parecida pero con prioridad dinámica.

En el tercer y último problema veremos una versión alternativa del **problema de los filósofos**, incorporando un cocinero a la ecuación. Esecialmente será una mezcla entre el problema de los fumadores y el estanquero y el de los filósofos en su versión inicial, aplicando una mezcla entre el primer y el segundo problema propuestos en este guión: el uso de varios tipos de monitores para tareas distintas unido al uso del mismo recurso de distinta forma.

1. **Se tiene un orfanato con n niños que comen y después han de lavarse los dientes. Para ello necesitan echarse pasta de dientes y cepillarse los dientes en el lavabo. Sin embargo en el orfanato hay unicamente cuatro botes de pasta de dientes y dos lavabos. Nunca puede haber más de un niño con un bote de pasta de dientes o en un lavabo. Resolver el problema de exclusión mutua relajada usando monitores.**

   Empecemos definiendo el proceso de cada niño:

```
process huerfano [i: 1..N]
var monitorPasta(4), monitorLavabo(2) : Monitor;
begin
        while true do begin
                Comer();                        // retraso aleatorio
                monitorPasta.Esperar();
                ServirsePasta();                // retraso aleatorio
                monitorPasta.Salir();
                monitorLavabo.Esperar();
                Cepillar();                     // retraso aleatorio
                monitorLavabo.Salir();
        end
end
```

   Para este problema necesitaremos dos monitores puesto que disponemos de dos recursos distintos. Si utilizásemos unicamente un monitor se darían exclusión mútua entre los lavabos y los botes de pasta de dientes, lo que no tiene sentido en este caso. Definamos ahora los dos monitores que utilizaremos:

```
class Monitor {
private:
        cond Cond;
        const int MAX;
```

```
        int en_uso;
public:
        Monitor (int max);
        void Esperar();
        void Salir();
};
```

Por último definamos los métodos del monitor:

```
Monitor::Monitor (int max) : MAX(max) {
        en_uso = 0;
}
```

```
void Monitor::Esperar() {
        if (en_uso == MAX) {
                Cond.wait();
        }
        en_uso++;
}
```

```
void Salir() {
        en_uso--;
        Cond.signal;
}
```

Podemos ver que a ambos recursos se accede de forma análoga (de hecho utilizan la misma clase) pero necesitan de dos monitores distintos para poder acceder simultáneamente a ellos.

2. **Tenemos un puente levadizo por el que quieren cruzar tanto coches como barcos. Sólo puede estar cruzando el puente o bien un barco o bien un coche de forma simultánea, y para ello el puente ha de estar en la posición correcta. Cuando un vehículo termina de cruzar el puente se estudia quien será el siguiente, teniendo en cuenta que:**

   ***a*) Si el puente está bajado tienen preferencia los coches, y viceversa.**

   ***b*) Si han pasado más de diez vehículos del mismo tipo seguidos la preferencia la tienen los vehículos del tipo contrario. Esta regla prevalece sobre la anterior.**

   Proporcionar una posible solución de este problema utilizando monitores.

   En primer lugar repasemos las condiciones del enunciado. Cuando hablamos de preferencia nos referiamos a justamente eso, preferencia. Es decir, si han pasado 15 coches seguidos pero no hay barcos que quieran cruzar el puente, seguirán cruzando coches hasta que llegue un barco. No tenemos por qué cambiar la posición del puente para volver a cambiarla acto seguido.

   Una vez aclarado esto presentamos una solución al problema. Para ello utilizaremos unicamente un monitor puesto que ha de darse exclusión mútua entre coche-coche, barco-barco y coche-barco. Tendremos un contador, *contador*, para saber cuantos vehículos del mismo tipo han pasado, aumentándolo en uno cada vez que circula un vehículo y reiniciándolo cuando cambiamos el puente de posición.

Por último la preferencia a la hora de cruzar el puente esta estudiada en el método $Monitor :: Signal()$, respetando las restricciones específicadas en el enunciado.

Declaremos ahora los procesos para coches y barcos, donde *monitor* será una variable compartida de tipo *Monitor*:

```
process coche [i: 1..N]
begin
        while true do begin
                LlegarAlPuente();                // retraso aleatorio
                monitor.EsperarPuenteAbajo();
                Cruzar();                        // retraso aleatorio
                monitor.Salir (true);
        end
end
```

```
process barco [i: 1..M]
begin
        while true do begin
        LlegarAlPuente();               // retraso aleatorio
                monitor.EsperarPuenteArriba();
                Cruzar();                        // retraso aleatorio
                monitor.Salir (false);
        end
end
```

La clase *Monitor* que regulará el transito sobre el puente será la siguiente:

```
class Monitor {
private:
        cond CondCoche, CondBarco;
        const int MAX = 10;
        int contador = 0;
        bool puente_abajo = true, cruzando = false;
        void Signal(bool preferencia_coche);
        void MoverPuente();
public:
        void EsperarPuenteAbajo();
        void EsperarPuenteArriba();
        void Salir(bool soy_coche);
};
```

Definamos los métodos del Monitor:

```
void Monitor::MoverPuente() {
        Mover();                // retraso aleatorio
        puente_abajo = !puente_abajo;
        contador = 0;
}
```

```
Monitor::Signal (bool preferencia_coche) {
        if (preferencia_coche) {
                if (CondCoche.queue()) {
                        CondCoche.signal();
                } else if (ConBarco.queue()) {
                        MoverPuente();
                        CondBarco.signal();
                }
        } else {
                if (CondBarco.queue()) {
                        CondBarco.signal();
                } else if(CondCoche.queue()) {
                        MoverPuente();
```

```
                        CondCoche.signal();
                }
        }
}
```

Nota: podemos ver que si ambas colas están vacías, *monitor.Signal()* no tiene ningún efecto.

```
void Monitor::EsperarPueteAbajo() {
        if (cruzando) {
                CondCoche.wait();
        }
        if (!puente_abajo) {
                MoverPuente();
        }
        cruzando = true;
}
```

```
Monitor::EsperarPuenteArriba (int max) : MAX(max) {
        if (cruzando) {
                CondBarco.wait();
        }
        if (puente_abajo) {
                MoverPuente();
        }
        cruzando = true;
}
```

```
void Salir(bool soy_coche) {
        cruzando = false;
        contador++;
        if (contador >= MAX) {
                soy_coche = !soy_coche;
        }
        Signal(soy_coche);
}
```

3. **El problema de los filósofos dice así: en una mesa redonda hay n filósofos, donde n es par. Entre cada dos filósofos hay unicamente un cubierto, ya sea cuchillo o tenedor (cada uno tendrá a un lado un cuchillo y al otro un tenedor). Sin embargo no se ponen de acuerdo y han de usar los cubiertos por turnos. Un filósofo habla durante un periodo de tiempo variable y acto seguido intenta comer, haciéndolo unicamente si ambos cubiertos están libres.**

   La versión planteada en este guión añade la siguiente modifiacion: **Hay un único cocinero haciendo platos para todos los filósofos. Cada filósofo come un plato distinto. Tras cocinar, el cocinero colocará una ración para un filósofo seleccionado de forma aleatoria. Si en el plato ya hay raciones, esta se añadirá al mismo sin ningún problema. Un filósofo puede comer unicamente si ambos cubiertos están libres y si hay comida en su plato, y comerá unicamente una ración. Tras esto suelta sus cubiertos y vuelve a hablar.**

   La mayor complicación de este problema es el hecho de que cada proceso/filósofo tiene que acceder a dos recursos compartidos, los cuales son independientes entre si. Esto significa que no podemos incluir ambos cubiertos en un mismo monitor.

Supongamos que lo hacemos, incluyendo los cubiertos del filósofo i en un monitor M. Es posible que el filósofo i no tenga comida pero los de sus lados si, $i-1$ e $i+1$. Por tanto ellos comenzarían a comer a la vez, accediendo simultáneamente al monitor M (cada uno usaría uno de los dos cubiertos). Imposible, pues están en un mismo monitor.

Resolveremos esto creando n monitores, uno para cada cubierto y tantos como filósofos. Representaremos el monitor número i como el cubierto de la izquierda del filósofo i, y el monitor $i+1$ como el de su derecha. En el caso especial del último filósofo, este tomará el último cubierto $(n-1)$ y el primero (0).

El problema ahora, de forma intuitiva, es que el proceso/filósofo i no puede acceder simultáneamente a los monitores i e $i+1$. Esto lo resolveremos mediante un sistema $Pedir-Tomar-Dejar$, en el cual esperamos a que ambos cubiertos estén libres $(Pedir)$, intentamos tomar ambos cubiertos $(Tomar)$ y los dejamos de nuevo en la mesa si no nos ha sido posible tomar ambos$(Dejar)$.

Por último la variante del cocinero la resolveremos añadiendo otros n monitores de otra clase nueva al problema. Es obvio que el acceso a la comida ha de ser de exclusión mutua ya que el filósofo decrementa y el concinero aumenta el contador de comida. Sin embargo podríamos pensar en incluir este contador dentro del monitor i ya definido, ahorrándonos así la definición de otra clase monitor. Esta solución sería errónea ya que no permitiría al filósofo i tomar un cubierto mientras el cocinero añade comida a su plato. O lo que es peor, no permitiría al filósofo $i-1$ tomar el cubierto i mientras el cocinero añade comida.

La comida se añade al sistema $Pedir-Tomar-Dejar$ de forma muy sencilla. Cada vez que tomemos los cubiertos comprobámos además si hay comida en el plato. El filósofo sólo comerá si ha obtenido ambos cubiertos con éxito y efectivamente tiene comida.

Tras esta explicación procedemos con la implementación. Para ello suponemos que los vectores $monitorComida$ y $monitorCubierto$ son variables globales de tipo $MonitorComida$ y $MonitorCubierto$ respectivamente y de tamaño n. La definición de los procesos es la siguiente:

```
process filosofo [i: 0..N-1]
var toma1, toma2, comida: boolean;
begin
        while true do begin
                Hablar();               // retraso aleatorio
                toma1 = monitorCubierto[i].Tomar();
                toma2 = monitorCubierto[(i+1) % N].Tomar();
                comida = monitorComida[i].HayComida();
                while (!toma1 || !toma2 || !comida) do begin
                        if (toma1) do monitorCubierto[i].Dejar();
                        if (toma2) do monitorCubierto[(i+1) % N].Dejar();
                        Hablar();       // retraso aleatorio
                        monitorCubierto[i].Pedir();
                        monitorCubierto[(i+1) % N].Pedir();
                        toma1 = monitorCubierto[i].Tomar();
                        toma2 = monitorCubierto[(i+1) % N].Tomar();
                        comida = monitorComida[i].HayComida();
```

```
                end
                monitorComida[i].Comer();
                monitorCubierto[i].Dejar();
                monitorCubierto[(i+1) % N].Dejar();
        end
end
```

```
process cocinero
var plato : integre;
begin
        while true do begin
                plato = Cocinar();      // retraso aleatorio, devuelve el
                    numero de plato
                monitorComida[plato].PonerRacion();
        end
end
```

Las clases de monitores que regularán la sincronización tendrán la siguiente definición:

```
class MonitorComida {
private:
        int contador_comida = 0;
public:
        void PonerRacion();
        bool HayComida();
        void Comer();
};
```

```
class MonitorCubierto {
private:
        cond Cond;
        bool cubierto_en_uso = false;
public:
        void Pedir();
        bool Tomar();
        void Dejar();
};
```

Implementación de los métodos de la clase *MonitorComida*:

```
void MonitorComida::PonerRacion {
        contador_comida++;
};
```

```
bool MonitorComida::HayComida {
        return contador_comida >= 0;
};
```

```
void MonitorComida::Comer {
        contador_comida--;
};
```

Implementación de los métodos de la clase *MonitorCubierto*:

```
void MonitorCubierto::Pedir {
        if (cubierto_en_uso) {
                Cond.wait();
        }
};
```

```
void MonitorCubierto::Tomar {
        bool aux = cubierto_en_uso;
        cubierto_en_uso = true;
        return aux;
};
```

```
void MonitorCubierto::Dejar {
        cubierto_en_uso = false;
        Cond.signal();
};
```

Por último cabe destacar que es imposible que se de interbloqueo: el cocinero está siempre en movimiento y los filósofos no mantienen los cubiertos en uso a no ser que estén comiendo.